2011

Gpu Based Lithography Simulation and Opc

Lokesh Subramany
University of Massachusetts Amherst

GPU BASED LITHOGRAPHY SIMULATION AND OPC

A Thesis Presented

by

LOKESH SUBRAMANY

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
Of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

SEPTEMBER 2011

Department of Electrical and Computer Engineering

GPU BASED LITHOGRAPHY SIMULATION AND OPC

A Thesis Presented

by

LOKESH SUBRAMANY

Approved as to style and content by:

_____
Sandip Kundu, Chair

_____
Russell Tessier, Member

_____ _____
Maciej Ciesielski, Member

_____
C.V. Hollot, Department Head
Department of Electrical and Computer
Engineering

ABSTRACT

GPU BASED LITHOGRAPHY SIMULATION AND OPC

SEPTEMBER 2011

LOKESH SUBRAMANY

B.E, E.C.E, VISHVESHWARIAH TECHNOLOGICAL UNIVERSITY

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Sandip Kundu

Optical Proximity Correction (OPC) is a part of a family of techniques called Resolution Enhancement Techniques (RET). These techniques are employed to increase the resolution of a lithography system and improve the quality of the printed pattern. The fidelity of the pattern is degraded due to the disparity between the wavelength of light used in optical lithography, and the required size of printed features. In order to improve the aerial image, the mask is modified. This process is called OPC, OPC is an iterative process where a mask shape is modified to decrease the disparity between the required and printed shapes. After each modification the chip is simulated again to quantify the effect of the change in the mask. Thus, lithography simulation is an integral part of OPC and a fast lithography simulator will definitely decrease the time required to perform OPC on an entire chip.

A lithography simulator which uses wavelets to compute the aerial image has previously been developed. In this thesis I extensively modify this simulator in order to execute it on a Graphics Processing Unit (GPU). This leads to a lithography simulator that is considerably faster than other lithography simulators and when used in OPC will

lead to drastically decreased runtimes. The other work presented in the proposal is a fast OPC tool which allows us to perform OPC on circuits faster than other tools. We further focus our attention on metrics like runtime, edge placement error and shot size and present schemes to improve these metrics.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Optical lithography is a step in the manufacture of Integrated Circuits (ICs) taking up to 30% of the time involved the manufacture of a chip [2]. In this process the features on a mask are transferred to the photoresist layer on a silicon wafer using ultraviolet light. Light from the source is passed through the condenser lens and is projected onto the mask. The diffraction pattern produced by the mask is captured by the projection lens and is focused onto the resist coated silicon wafer. The photoresist is activated by the incident light and undergoes chemical change. The photoresist is then etched away by a chemical etchant leaving behind the mask features on the silicon wafer. The lithography system is shown in Figure 1.

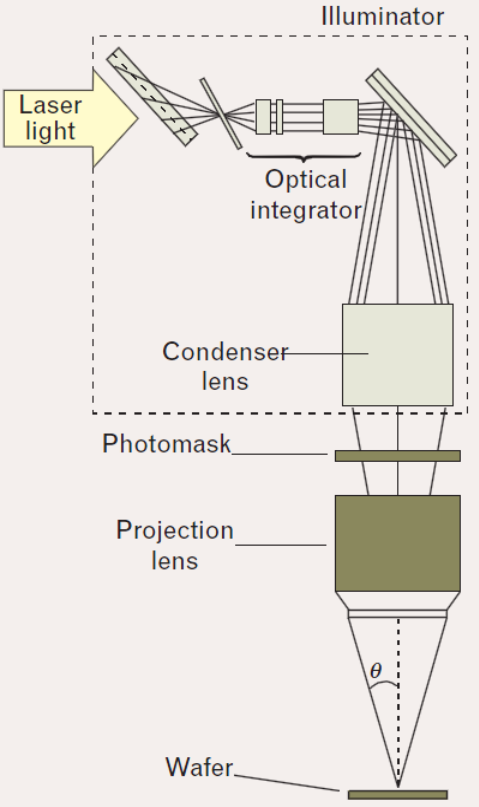

Figure 1. Lithography System [1]

The light source used in the lithography process today has a wavelength of 193 nm. With this light source, devices having critical dimension of 45nm, 32nm and 25nm are being manufactured. This disparity between the feature size and the source wavelength is shown in Figure 2. The improvements in optical lithography have slowed down due to the absence of suitable sources of low wavelength. There are inherent difficulties in printing feature sizes below the wavelength of light, called sub-wavelength lithography. These difficulties lead to degradation of the printed pattern compared to the source mask. Hence, sub wavelength lithography relies on a set of resolution enhancement techniques (RET) such as off-axis illumination, phase-shift masking, layout constraints and optical proximity correction (OPC) to improve the quality of the printed pattern.

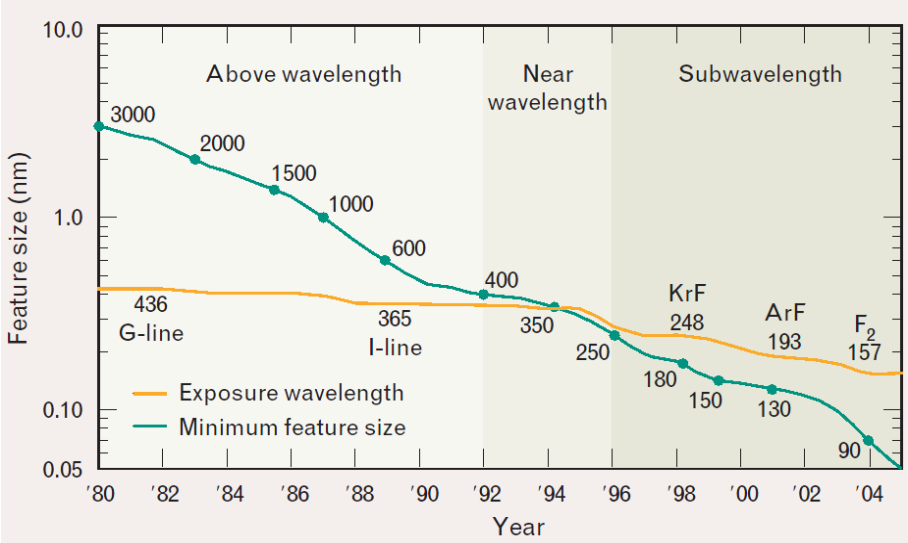

Figure 2. Decrease in feature size and source wavelength [1]

A lithography simulator is used to obtain the aerial image intensity on the surface of the photoresist. A resist model is used to model the etching process and to identify the final pattern after etching. The simulator can be used to troubleshoot problems in the fab reducing the number of test wafers [2], as an aid in design routing [3], and it also allows us to improve the quality of the printed pattern by its use in OPC [4][5].

In OPC the goal is to improve the quality of the printed pattern by making changes to the original mask. The mask is modified to compensate for effects that occur during the lithography process, leading to an improved wafer pattern. OPC is an iterative process in which small changes are made to the mask and the effect of these changes is observed by lithography simulation. Thus, lithography simulation becomes a part of a feedback system and the need for a fast lithography simulator cannot be overstated. OPC allows us to attain a higher yield for a given minimum feature size, improves the performance of a given minimum feature size and allows us to use smaller design rules [5]. An example of OPC is given in Figure 3.

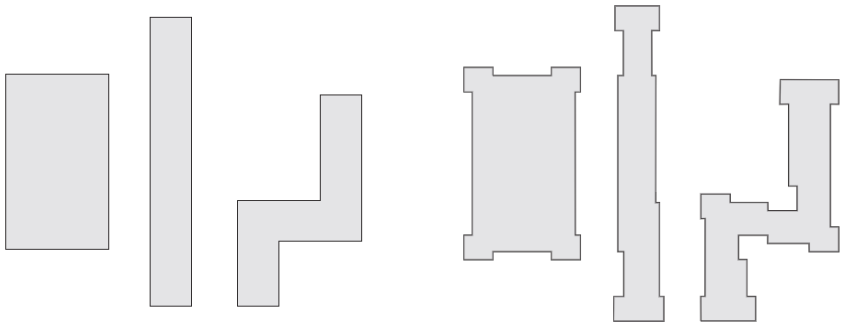

Figure 3. An example of OPC [5]

For certain computations GPUs can exhibit higher computational power compared to CPUs of contemporary generation; this can be seen in Figure 4. The reason for this disparity in performance can be attributed to the difference in the design philosophies and the applications for which the respective devices were designed. GPUs were primarily designed to render graphics for animation movies, and CAD modeling. But the game industry has been a primary factor in driving performance in GPUs. Games require massive amounts of floating point computations in every frame, and a constant frame rate must be maintained [6].

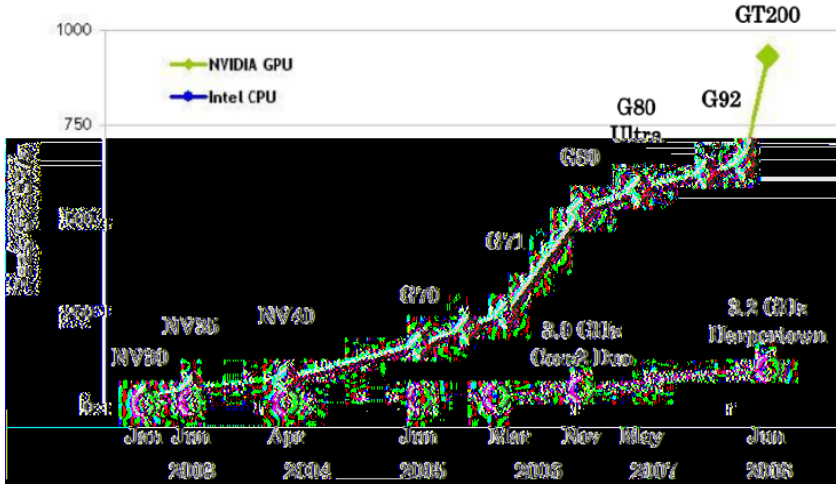

Figure 4. Performance disparities between CPU and GPU [7]

The games industry is always trying to improve the graphical fidelity of games which requires increased computation and this in turn creates demand for more powerful

GPUs. This has caused the GPU manufacturers to optimize GPUs for high throughput and a large memory bandwidth. Thus, the CPU is optimized to run a single thread efficiently with memory latency being minimized with the help of a large cache, while a GPU is optimized to run a large number of threads. The memory latency is amortized over these threads with the help of a large memory bandwidth. These differences are highlighted in Figure 5.

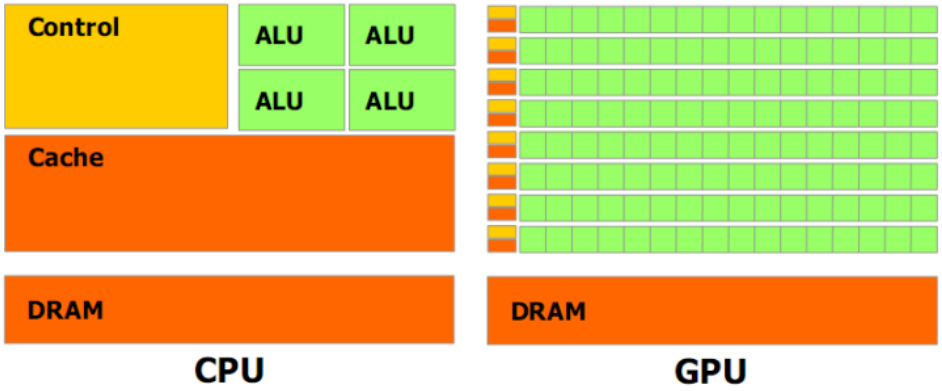

Figure 5. Architectural Differences between CPU and GPU [7]

There are a few applications which are able to use the available processing power and bandwidth of GPUs to accelerate their execution. In [8] the authors look at using GPUs for physical design automation, while in [9] the authors discuss about how GPUs can be used as a general computation resource. More information about GPUs and their use in solving non-graphical problems can be found in [10]. By implementing the lithography simulator on a GPU we gain a fast simulator which can be used to perform OPC faster than other implementations.

1.1 Thesis outline

The outline of this thesis is as follows, Chapter 2 describes the background and related work which includes lithography using wavelets, and a section on OPC. Chapter 3

discusses the GPU architecture and its programming. Chapter 4 deals with the implementation of the lithography simulator on a GPU and in Chapter 5 the implementation of OPC using wavelets and various improvements to the basic OPC method are detailed. Chapter 6 presents the experimental results of the methods described in Chapter 5 followed by the conclusion of the thesis in Chapter 7.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Optical Lithography Simulation [2]

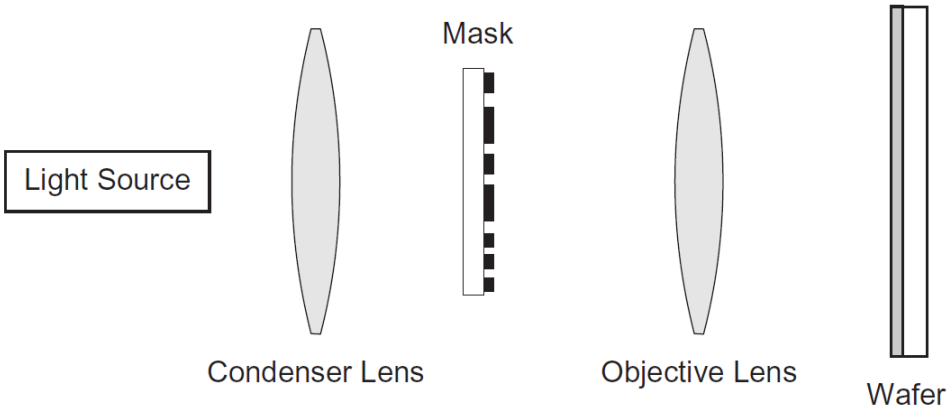

Figure 6. Generic lithography system [2]

A lithography system consists of a source, a condenser lens system, a mask, objective/projection lens system and a resist coated wafer, as shown in Figure 6. The source must be powerful enough to project the mask pattern onto the wafer; the mask consists of transparent glass etched with the circuit pattern. The light passes through the mask and gets diffracted. This diffraction pattern is captured by the objective lens and is projected onto the photosensitive resist.

2.1.1 Aerial Image Formation

The diffraction of light can be explained by Huygens' principle, where the optical wave front can be thought to be made up of point sources of light. When light passes through a slit the wave fronts begin to diverge from the slit leading to spreading of the light beam. If the distance between the objective lens and the mask is large, then it is

termed Fresnel diffraction. Commercial lithography systems satisfy the Fresnel diffraction condition.

The mask is described in terms of a mask transmittance function $t_m(x, y)$ where the transmittance is 1 for a clear region and 0 for the chrome/opaque region. The plane x'- y' describes the entrance to the objective lens which is the diffraction plane and z is the distance between the wafer and the objective lens. The light is monochromatic having a wavelength of λ, and the refractive index of the medium is n. f_x and f_y are scaled coordinates given by $f_x = nx'/(z\lambda)$, $f_y = ny'/(z\lambda)$. For a given mask the electric field of the diffraction pattern is given by the Fraunhofer diffraction integral

$$T_m\left(f_x, f_y\right) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} E_i(x,y)t_m(x,y)e^{-2\pi i(f_x x + f_y y)}dxdy$$

Here E_i is the electric field incident on the mask.

This equation is a Fourier transform which implies that the diffraction pattern is the Fourier transform of the mask pattern transmittance. The diffraction extends on the x' – y' plane, however due to the limited size of the objective lens all the diffraction orders are not captured. Only the orders that fall within the aperture of the lens form the image. The size of the lens is described by a term called the numerical aperture which is defined as the sine of the maximum half angle of light that can enter the lens times the refraction index of the surrounding medium.

$$NA = n\ sin\theta_{max}$$

If the numerical aperture is large, then more orders of diffraction can be captured leading to a better image. To create a reasonable image, at least the zero order and the first orders of diffraction need to be captured. The theoretical resolution of this system is given by the equation

$$R = k_1 \frac{\lambda}{NA}$$

where k_1 is a parameter that depends on the lens system.

Theoretical resolution describes the smallest pitch that can be imaged using the lens system for normally incident plane waves.

As the diffraction pattern is the Fourier transform of the mask, the mask pattern can be recreated if the objective lens performs an inverse Fourier transform operation on the diffraction pattern. So we define a parameter called the Pupil function P, this function is the transmittance of the lens from the entrance pupil of the lens to the exit pupil. It describes the portion of light that makes its way through the lens and is given by

$$P(f_x, f_y) = \begin{cases} 1, & \sqrt{f_x^2 + f_y^2} < \frac{NA}{\lambda} \\ 0, & \sqrt{f_x^2 + f_y^2} > \frac{NA}{\lambda} \end{cases}$$

The pupil function is 1 inside the aperture and 0 outside. The product of the pupil function and the diffraction pattern gives us the light that exits the objective lens. Thus, the electric field at the wafer plane is given by

$$E(x, y) = F^{-1}\{T_m(f_x, f_y)P(f_x, f_y)\}$$

Where F^{-1} represents the inverse Fourier transform. The aerial image is defined as the intensity distribution in air at the wafer plane and is the square of the magnitude of the electric field.

2.1.2 Resist Model

The aerial image is formed on the surface of the resist. The resist is activated by the light and undergoes a chemical change. The resist must now be etched by chemical means to obtain the required pattern. In wet etching a chemical etching agent is used to remove the film from under the non-activated photoresist. Then the activated photoresist

is removed by another chemical agent. At the end of this process the mask pattern is transferred to the wafer.

In order to obtain the shape of the pattern after etching in lithography simulation, the effect of incident light on the photoresist also needs to be modeled. The pattern produced on the photoresist depends on the exposure time and the dosage of light. As exposure time can be controlled accurately it is always considered to be nominal and hence is not a factor in the resist model, while dosage can vary temporally and contributes to process variation. In [14] the author goes into more detail about the effect of light on the photoresist. There are two resist models, variable threshold resist model and constant threshold resist model.

In constant threshold model a single intensity value is calculated based on a fraction of the difference of minimum and maximum intensity of the aerial image. All the points which have this intensity value will lie on the edge of the final pattern. Although simplistic this method provides good results and is also computationally less intensive. We use this model in our work. A constant threshold model is presented in [13]; in this model a constant value obtained from normalized aerial image intensity is used as the threshold value for the photoresist. This is also known as the 0.3 contour method. Understandably, compared to the variable threshold model, the constant threshold model is not as accurate, but has the advantage of being less intensive computationally, and is considered to be good enough

In variable threshold resist model, a function is used to determine the threshold for activation of the resist. This value is then applied to a small area. Randall et al. describes this process in more detail [15]. In the Variable threshold resist model presented in [5], a data dependent threshold is used to determine at which normalized light intensity the printed edge will appear [5]. This model is obtained from empirical measurements.

2.1.3 Previous Work

Various models for lithography have been presented over the years. A method that simulates the mask diffraction by using the finite difference time domain (FDTD) method on the electromagnetic equations is presented in [12]. But this method is very time intensive and needs copious computational resources. In [10] , the authors use multi-resolution time domain method (MRTD) in order to speed up this process. In [5] to speed up aerial image simulation, Cobb used decomposition of Hopkins partially coherent equations. As this method also needs large amount of computational resources, in [13], the authors use rectangle look-up to speed up this simulation.

Recently in [16], the authors have used wavelet transform to generate the aerial image. Using the theory of single slit diffraction pattern where the image resembles a $sinc^2$ function, the aerial image is obtained by applying the wavelet to the entire mask. This approach aims to speed-up simulation using wavelet transform. This has been extended to a mask containing multiple polygons where a 2D $sinc^2$ pulse is convolved with the mask to generate an aerial image. It has been shown that the aerial image obtained by using the 2D $Sinc^2$ pulse, coupled with a constant threshold resist model, closely approximates the aerial image obtained by commercial lithography tools [16].

2.2 Wavelet Transform [17]

Wavelet transform is similar to Fourier transform and is used to analyze signals that are aperiodic, noisy and intermittent. This method allows us to analyze a signal simultaneously in both time and frequency. The equation for a wavelet transform is given below [17].

$$T(a,b) = w(a)\int_{-\infty}^{+\infty} x(t)\Psi\left(\frac{t-b}{a}\right)dt \qquad (1)$$

In the equation, x (t) represents the mask, while ψ represents the wavelet which in our case is a $sinc^2$ pulse. 'a' is a parameter which specifies the scale of a wavelet while 'b' is the translation parameter which specifies the temporal location of the wavelet. w (a) is a weighting function set to 1/sqrt (a) for reasons of energy conservation. T (a, b) is the transform value at scale 'a' and location 'b'.

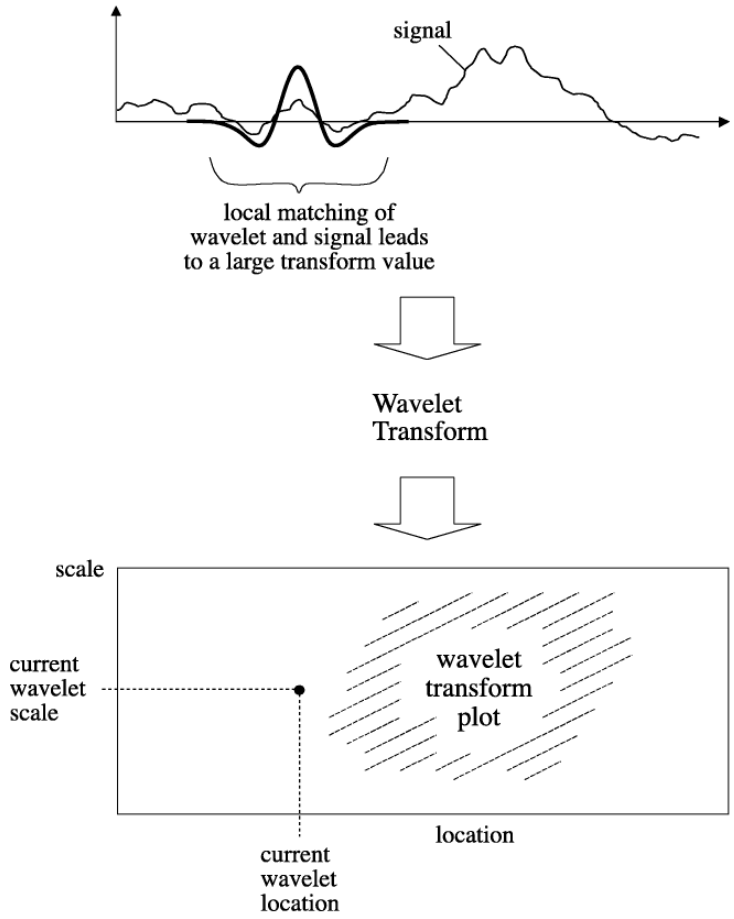

Figure 7. 1D wavelet transform [17]

Figure 7 shows the 1D wavelet transform operation. The scale of the wavelet is fixed and then the wavelet is translated in time to obtain the transform value. This process is then repeated with a different scale of the wavelet. On doing so for various scales we derive the wavelet transform plot. The wavelet having a scale of a = 1 and translation parameter b = 0 is called the mother wavelet. By changing the values of 'a' and 'b' we

obtain daughter wavelets. For lithography simulation we use the $Sinc^2$ pulse as the wavelet. The scale of the wavelet is fixed as changing the scale changes the defocus value of the system. The $Sinc^2$ pulse is the image pattern obtained when light is shone on a slit. As the lithography mask can be imagined to be an integration of succeeding slits, we can obtain the aerial image by using the $Sinc^2$ wavelet pulse. For a 2D mask we use a 2D $Sinc^2$ pulse as shown in Figure 8.

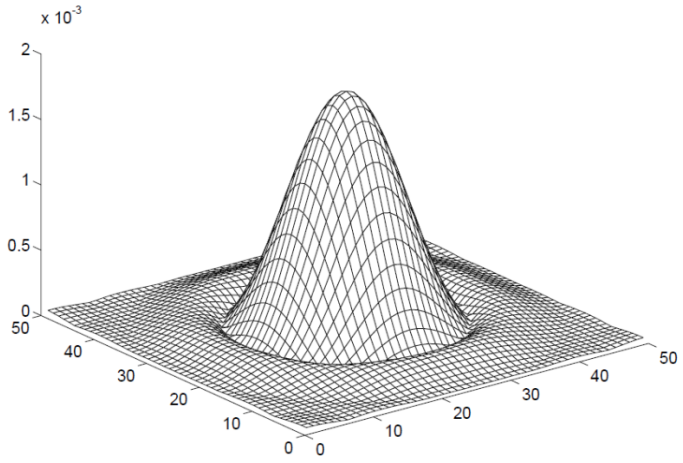

Figure 8. A 2D $Sinc^2$ pulse [5]

2.3 Lithography simulation using wavelets

In this section we provide a brief overview of the CPU implementation of wavelet based lithography simulation presented in [16]. To calculate the aerial image using wavelets we need the mask description and the wavelet. We consider the use of binary masks, in which the presence of a shape indicates a zero transmittance and has a value of 0; a clear area indicates 100% transmittance and hence has a value of 1. So points within a contour (a solid shape on the mask) will have a lower intensity value compared to points outside a contour.

A circuit mask can be fractured into rectangles which are linked together to form various contours. On a metal mask layer, the contours can be visualized as a chain of rectangles that create electrical connections between various devices on the silicon wafer.

The edges of the rectangles are lines and these lines are the basis of simulation points. So in order to calculate intensity we only need to add up all the light that makes its way to the point under consideration. To find the entire aerial image for a chip we need to repeat this calculation for all points on the mask.

We only need to find where the outer edge of a feature/contour lies and are not concerned about the intensity values within the contour. So we select certain simulation points only along the edge of the contours. The number of simulation points can be reduced further by recognizing the fact that these points need not be uniformly distributed. In [16] the simulation points are generated based on the size of the contour, the proximity of the contour to other contours and the number of corners in the contour. By judiciously selecting simulation points we can minimize the loss in accuracy and gain in simulation performance.

Once all the simulation points have been determined, the calculation of image intensity ensues. On a mask the aerial image intensity at a point depends on the features that lie in its optical diameter (~1μm) [5]. This optical diameter is also referred to as a tile. To obtain the aerial image intensity on the die, we transform the mask description with the wavelet function, where the wavelet function is defined only in the optical region of influence. This is shown in Figure 9. In order to implement this process, we create a mask tile, which contains a mapping of mask contours in the optical region in a 2D matrix. The matrix contains 1s and 0s corresponding to the contours in the mask. The wavelet tile is also a matrix containing values of the wavelet. To find the wavelet transform we multiply the corresponding elements of the two matrices and sum all the products to arrive at a single value, which is the aerial image intensity.

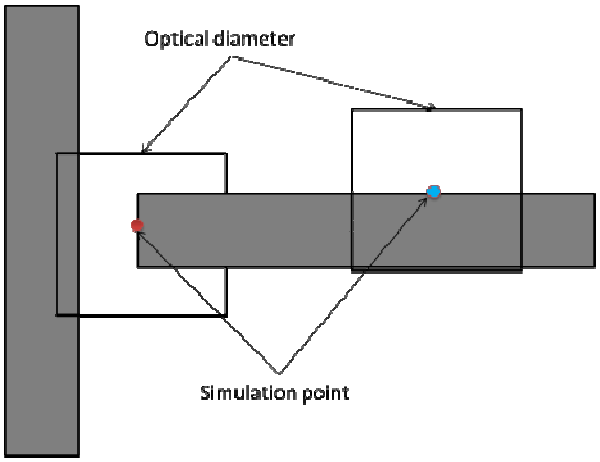

Figure 9. Optical diameter and simulation points

We use a constant threshold model to calculate the image intensity which defines the edge. This edge intensity is calculated by the following formula

$$edge\ intensity\ value = 0.3 * (max - min) + min \qquad (2)$$

where max and min correspond to the maximum and minimum intensity values in the mask.

The intensity value of every simulation point is computed and if this value is greater than the value obtained from equation (2), the location of the simulation point is moved and the intensity recalculated. This process is repeated until a location is found whose intensity is less that the aerial image intensity. This point now is a part of an edge. On repeating this process for all simulation points for a given edge of the mask, the aerial image edge can be found. This can be seen in Figure 10 where the points in blue represent the aerial image which is also the final location of the simulation points, and the points in green represent the original mask contour.

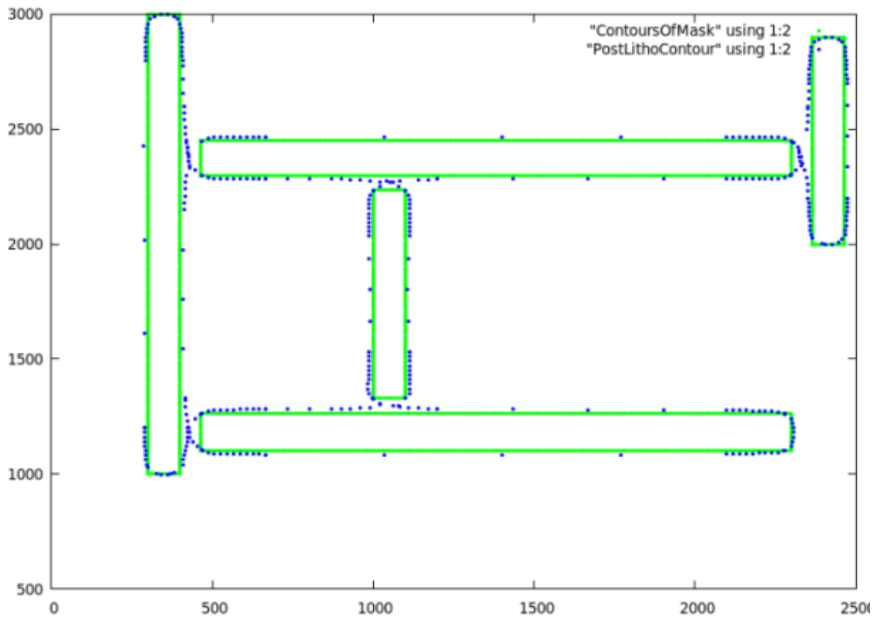

Figure 10. The original mask and the aerial image using the method described in [16]

2.4 Optical Proximity Correction

The line width of the pattern printed on the silicon wafer varies as a function of the proximity of nearby features. An isolated line will be printed wider than a dense line. This is a result of a fundamental limitation in the optics used in lithography. This difference between the desired and actual printed pattern on the wafer is a systematic error and it should be possible to correct for this error. The correction is carried out by changing the feature on the mask to compensate for the proximity effects which is called optical proximity correction. So the goal of OPC is to obtain the optimal mask to get the desired pattern on the resist. This is often called the 'inverse problem' in imaging. OPC can be categorized into Rule based and Model based, the following sections explain each of these approaches in further detail.

2.4.1Rule based OPC

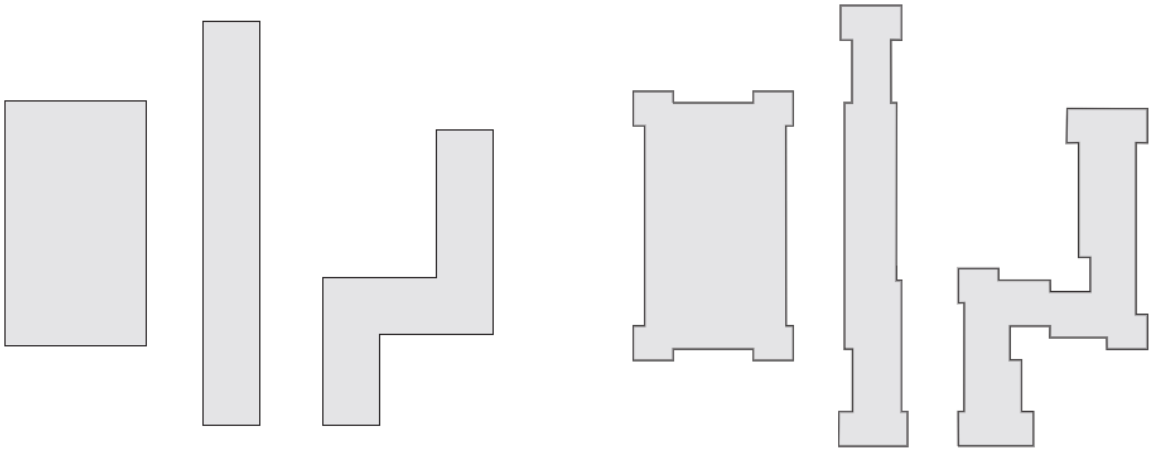

Figure 11. Rule based OPC [2]

In rule based OPC a set of rules are created and a correction pattern is created for each of those rules. The entire mask is searched for patterns which match the rules and if found the correction is applied to the pattern. Rule based OPC is simple to implement for one dimensional correction but can get very complicated for two dimensional effects like corner rounding and line end shortening [2]. An intermediate approach is to use a separate set of rules for these effects and use another set of rules for 1D edges which is called 1.5D correction. An example of rule based OPC is shown in Figure 11, the pattern on the left is the original mask pattern while the one on the right is the pattern after OPC.

Although implementing the rule based system is conceptually simple, the rules and the corrections for the patterns must be experimentally determined. The rules are limited to a specific lithography process and must be regenerated if any of the optical parameters change. A small increase in accuracy leads to a large increase in the number of rules, and at process nodes lower than 130nm the required accuracy increases. Rule based OPC was used extensively until 250nm; however by the 130nm node the accuracy and robustness of rules based OPC decreased [2].

2.4.2 Model based OPC [2]

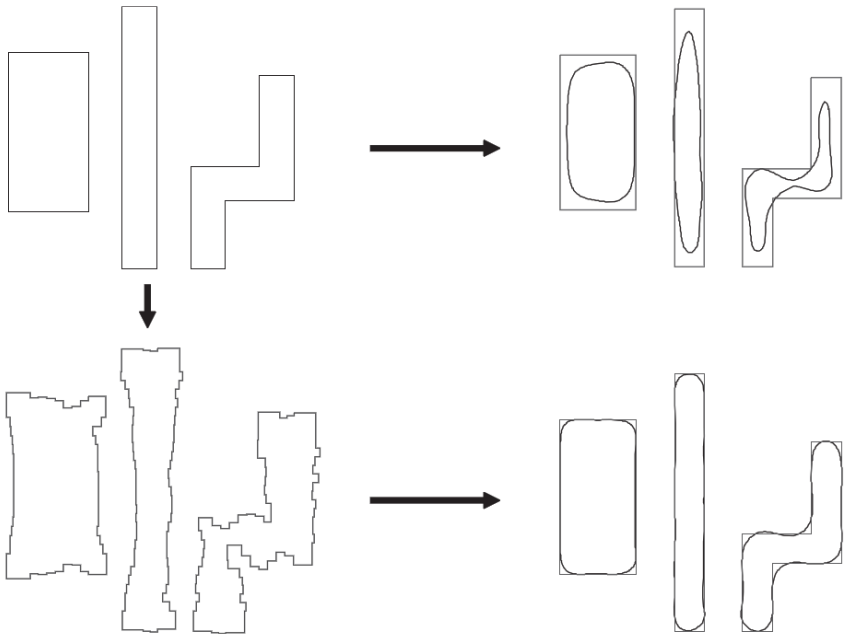

Figure 12. Model based OPC [2]

Model based OPC replaced rule based OPC as the process node decreased to 90nm. In model based OPC a lithographic model is used to derive the aerial image of the chip. The proximity effects are taken into account during the simulation of the mask pattern which leads to the aerial image. Once the aerial image is obtained the edges of the features on the mask are iteratively moved until the aerial image shape closely matches the desired shape. In this method considerable effort is spent on refining the lithographic model. As a good model should be able to simulate full chip masks containing millions to billions of features, it should be highly parallelizable and accurate. Figure 12 shows an example of model based OPC; the first pattern is the original mask pattern while the pattern on the right is the aerial image after lithography simulation. The second pattern is corrected pattern and the corresponding aerial image.

In model based OPC, the mask is divided into edges. Each edge can be independently moved. The mask is simulated and the aerial image is obtained on the

photoresist. The aerial image is compared to the original mask and an Edge Placement Error (EPE) is calculated. This parameter can be used as a metric for the quality of OPC. If the EPE is low then the aerial image is close to the desired shape. The edges are now moved iteratively and resulting pattern is simulated again to get a new aerial image and a new EPE. This process is carried out until the EPE attains an acceptable value. In order to reduce the mask complexity, the edge positions are snapped to a grid. The aggressiveness of OPC can be controlled by controlling the minimum size of the edge. A more aggressive OPC results in more fragments which increases the mask cost.

2.4.3 Previous Work

The field of OPC is quite mature and there have been several implementations. The early work in model based OPC was performed by Rieger et al [19] [20] [21]. Cobb et al [5] have implemented an OPC algorithm based on EPE. This approach was improved upon by the use of a Mask Error Enhancement Matrix (MEEM) in [18]. Other works have focused on decreasing the runtime of various OPC implementations; in [22] the authors present a new convergence scheme which decreases the number of iterations while in [23] the authors use a neural network to speed up OPC. A GPU based implementation using Hopkins sum of coherent sources approach to derive the aerial image has been presented in [25] while a hardware accelerated implementation is presented in [26].

CHAPTER 3

GPU ARCHITECTURE AND PROGRAMMING

Increasing the clock speed of a single core is becoming infeasible because of the large increase in dissipated power, and also higher clocks push the boundary of the switching speed of the transistors. This has led to the end of the clock speed wars of the Pentium era and to the core wars of the current generation, where the CPU manufacturers like Intel and AMD are adding more and more cores in succeeding generation of CPUs. Thus, the future of computing lies in parallelism and only multithreaded code can take advantage of the available computing resources and exhibit performance gains when moving from one generation to another.

Graphics Processing Units can be found in most of the computers today where they are used to render images onto screens. About six years ago they were fixed in their function and were suitable only for running 3D applications. Since then they have become increasingly programmable. The changes have been as a result of modifications is hardware as well as application programming interfaces [19]. More information about the GPU architecture and the recent changes in the architecture which make it amenable to general purpose computing can be found in [19]. This notion of using GPUs for non-graphics applications is called General purpose computation on GPU (GPGPU). The GPU was designed for a set of applications that have the following characteristics, the computational requirements are large, there is substantial parallelism and throughput is more important that latency [19].

Although GPUs have always had an edge over the CPUs in terms of theoretical computational power, general applications cannot make use of this available power due

to limitation inherent in the GPU. The reason for this disparity in performance can be attributed to the differences in the fundamental design philosophies between the CPU and the GPU. The design of a CPU is optimized for sequential code performance; a lot of logic is devoted to allow instructions from a single thread of execution to execute in parallel or out of order while maintaining the appearance of sequential execution. Large caches are provided to hide the instruction and the data access latencies.

Memory bandwidth is also another important issue. Graphics chips have about 10x the bandwidth of the available CPUs. Usually the bandwidth between the CPU and the main memory is around 15Gb/s, while the latest GPUs have about 100Gb/s of available bandwidth. But the bandwidth between the main memory and the GPU is about 8Gb/s, so you pay a penalty while transferring the data to and from the GPU. [29] Compares the latency and the bandwidth between the CPU and main memory and a GPU and its global memory.

The architecture of the GPUs is governed by the needs of the fast growing video game industry. There is a tremendous pressure for to perform a massive number of floating-point calculations in each frame in advanced games. This demand pushes the GPU vendors to look for ways to maximize the chip area that is dedicated to floating-point calculations. The general philosophy for GPU design is to optimize for the execution of massive number of threads. The hardware spawns a large number of execution threads to find work to do when some of them are waiting for long-latency memory accesses, minimizing the control logic required for each execution thread. Small cache memories are provided to help control the bandwidth requirements of these applications so that multiple threads that access the same memory data do not need to all go to the DRAM. As a result, much more chip area is dedicated to the floating-point calculations. CUDA (Compute Unified Device Architecture) provides a C like

programming paradigm which allows us to harness the computational resources of the GPU.

It should be clear now that GPU is designed as a numeric computing engine and it will not perform well on some tasks that CPUs are designed to perform well. For example, due to the limited cache present in the GPU, branch heavy code will face a huge penalty in execution on the GPU. Therefore, one should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numeric intensive parts on the GPUs. This is why the CUDA programming model is designed to support joint CPU-GPU execution of an application. We look at the CUDA programming model after a brief introduction to the architecture of a GPU.

3.1 GPU Architecture

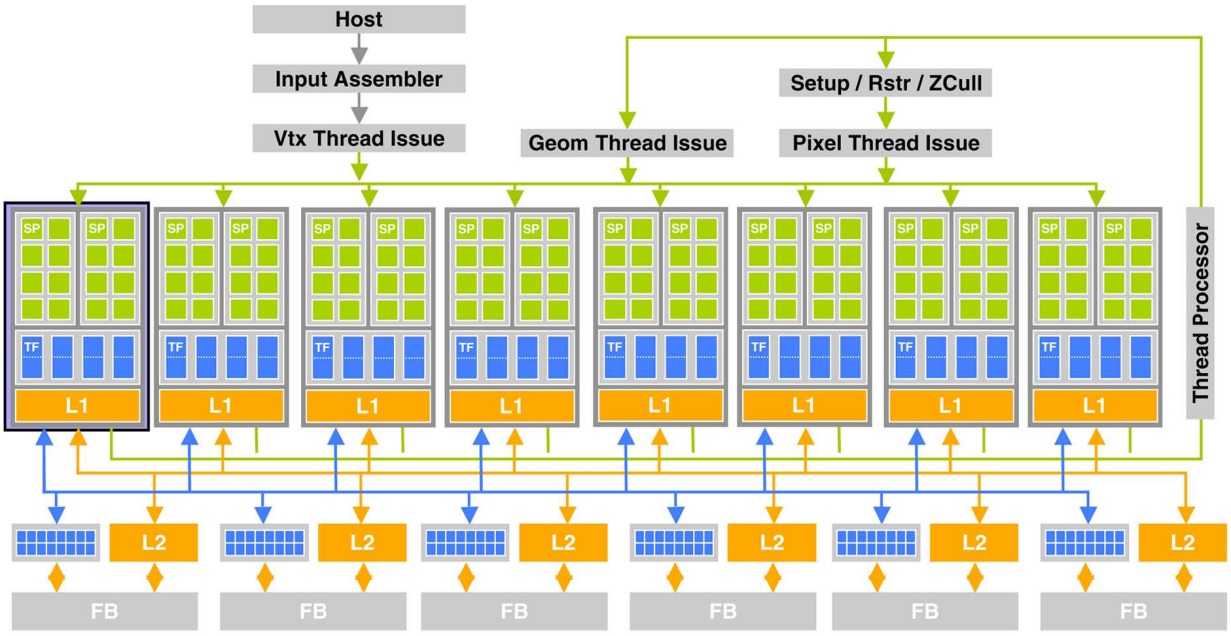

Figure 13. GPU architecture

Figure 13 shows the architecture of a modern GPU. It is divided into 16 Streaming Multiprocessors (SM). Two SM form a block. Each SM in turn consists of 8 Streaming Processors (SP) giving us a total of 128 SP. In the GPU used for my thesis (Tesla C870) the SP runs at 0.92 GHz. Each SP has a Multiply and Add (MAD) unit and an additional multiply unit. In addition to those units we also have units that perform SQRT, Sin, Cos operations. A SP is shown in Figure 14.

The GPU has about 1.5GB of memory. This memory is divided into global memory, constant memory, registers, shared memory and texture memory. The host can write to and read from the global and constant memory. Constant memory allows read only access by the device and provides faster and more parallel data access paths for the kernel execution compared to global memory. Currently, the total size of constant memory is limited to 65KB. Each SM also has a limited amount of cache.

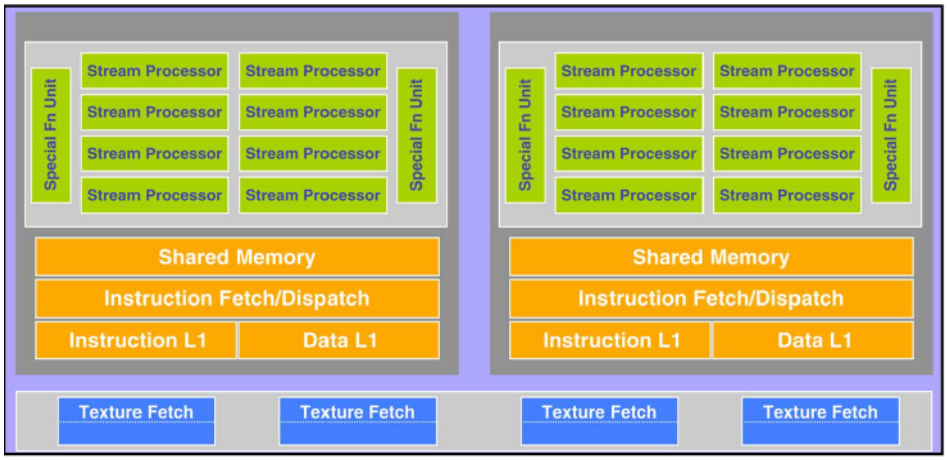

Figure 14. A pair of Streaming Processors

Registers are allocated to individual threads and are used to store frequently accessed private variables. Threads cannot share the data in the registers among themselves. Shared memories are allocated to thread blocks. All the threads in a block can read from and write to this memory. In the device used for my thesis, each thread

block has access to about 16KB of shared memory. Data in these memories have very low access times and have more parallel paths compared to the global and constant memories.

3.2 Programming a GPU using CUDA

There are one or more phases in a CUDA program, the phase that has a large amount of parallelism is executed on the GPU while the code that has little parallelism is executed on the CPU. To the CUDA programmer the CPU is the host and the GPU is the device that accelerates functions having a large amount of parallelism. In a typical CUDA program, the CPU starts the execution, before the GPU is used for computation, the data must be copied from the main memory to the GPU memory. When required the CPU invokes the kernel function.

When the kernel function is invoked the execution is switched to the GPU. The kernel function generates a large number of threads to take advantage of the multiple processing units in the GPU. This collection of threads is called a grid. When the kernel completes its execution, the grid terminates and control is returned to the CPU.

3.2.1 CUDA threads

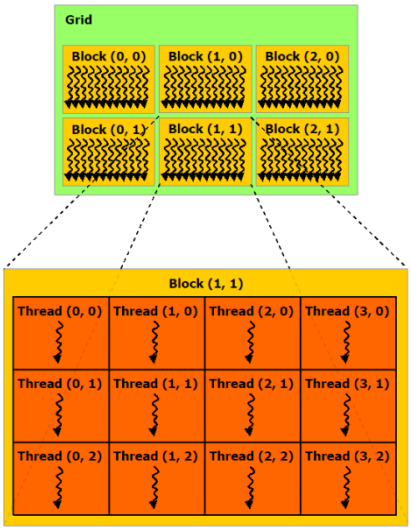

Figure 15. A grid and a block of threads [7]

All threads in a grid are identical and are organized into two levels as shown in Figure 15; each level has ids assigned to the threads by the CUDA runtime. The lower level id is the thread id which is represented by the built-in variable threadIdx. This variable is a three component vector and can be used to identify a thread in each dimension using threadIdx.x, threadIdx.y and threadIdx.z variables. The threads are grouped into thread blocks and the blocks in turn are laid out in two dimensions. Similar to threadIdx, blockIdx is also a three component vector that can be used to select a block. So in order to identify a single thread, we need to generate its index based on the number of blocks in the thread, the block index, and the index of the thread in the block.

$$thread\_idx = blockIdx.x * blockDim.x + threadIdx.x ;$$

The number of threads in each dimension of a block as well as the number of blocks in a grid can be specified at runtime. The number of threads in a block is limited to 512; these threads can be distributed in 3 dimensions in any fashion. The number of threads and blocks are specified as parameters of the kernel at runtime. These variables

are defined as *dim3* type which is a *struct* with three fields. An example of a configuration is shown below. The first statement sets up the block configuration while the second statement sets up the grid configuration. The third statement is the kernel launch.

```
dim3  blockDimension(4, 4, 4);

dim3  gridDimension(5,2,1);

        kernel<<< blockDimension, gridDimension>>>(….);
```

The threads in the same block can synchronize their execution and also communicate via shared memory. Barrier synchronization can be used to synchronize the threads in a block, in barrier synchronization; all threads will be stopped at the point where the function was called. Only after all the threads have reached that point will execution continue. The threads of a block are assigned to the same unit for execution to minimize the waiting times. The threads from different blocks cannot synchronize with each other.

The CUDA run time system does not guarantee the order of execution of thread blocks. This means that there cannot be any dependencies between thread blocks. This condition is necessary to aid scalability. The number of execution units in a GPU can vary dramatically depending on the market segment the particular GPU is targeted for. Some GPU have 128 units other 64, 512 and so on. By allowing the device to schedule the execution of a block at any time the run time environment can take advantage of all available units. When the code is executed on a device having a large number of SPs, more thread blocks can be executed simultaneously and less blocks on a device having fewer execution units. As the blocks are not dependent on each other this will not pose

any problems and the same code, without any modification will execute faster on more capable hardware. The scalability issue is demonstrated in Figure 16.

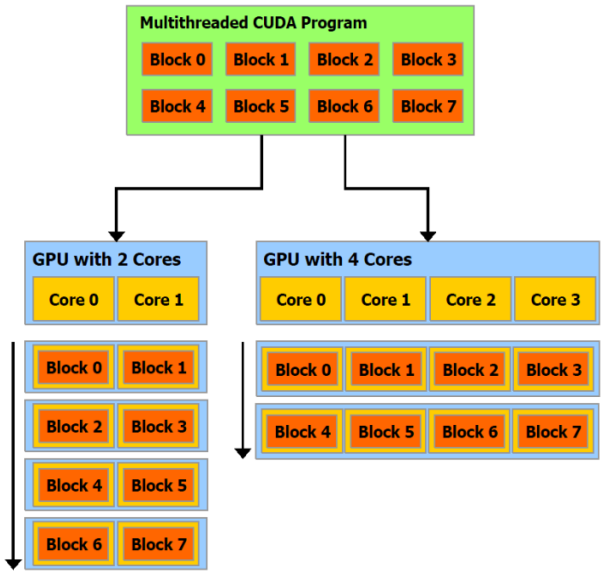

Figure 16. GPU scalability [7]

Once a block of threads is assigned to a Streaming Multiprocessor for execution, the threads are further dived into Warps. On the Tesla C870 a warp has 32 threads with threads having consecutive thread id values. The number of threads in a warp differs for different devices. At any point of time only one of the warps is being executed. The Tesla C870 device can have 24 warps residing in the SM at any point of time. When the instructions from one warp are waiting on the results, these instructions are replaced by instructions from another warp. The latency of an instruction is successfully hidden by scheduling and executing instructions from another warp. The warp scheduling incurs zero penalties as there is hardware support available for scheduling. Hardware support for thread switching allows greater flexibility in the implementation of our algorithm. As the overhead of creating and switching of threads is very low, a large number of threads can

be created which work independently on various parts of the circuit. This also allows us to hide memory latency as a thread which is waiting on a memory access can be quickly switched for a thread which has its data already available for computation. These factors further reduce runtime on the GPU.

3.2.2 CUDA Memory model

The memory on the device is divided into Global memory, Constant memory, Texture memory, Shared memory, Registers and Cache. The data from the CPU and the main memory can be transferred to the global memory, constant memory or the texture memory. The GPU has only read access to the constant memory and texture memory while it can read and write data onto the global memory. The constant memory allows faster and more parallel accesses to the kernel. Shared memories are local to a thread block and only threads within a block can access this memory. So inter thread communication within a block can be carried out by using shared memory. Registers are allocated to threads and are used to store frequently accessed variables that are local to each thread.

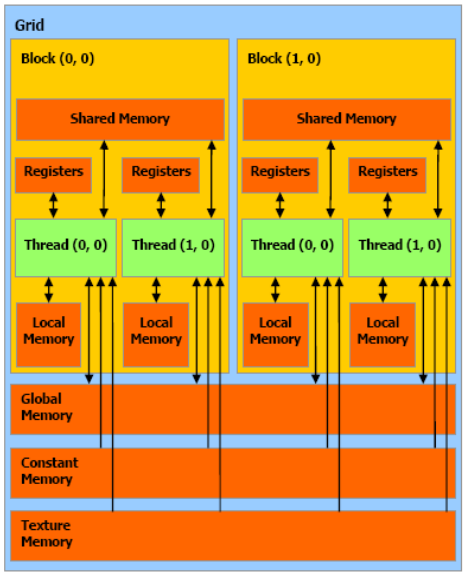

Figure 17. GPU memory [7]

If a variable declaration is preceded by the "__shared__" keyword, it declares a shared variable; the scope of this variable is limited to the thread block and must reside within a kernel or a device function. A private copy of this variable is created for all thread blocks and this variable is destroyed only when the kernel terminates its execution. In the TeslaC870, the shared memory is limited to 16KB per SM. Constant variable are declared with the "__constant__" keyword. These variables must be declared outside the function body. The scope of this variable is the entire grid that is all the threads in the grid will have access to this variable. This variable is destroyed only when the entire application is terminated. These variables are stored in global memory but are cached. The total size of the constant memory is limited to 65KB. The variables placed in global memory are visible to all the threads in the kernel. Accesses to global memory are slow and these variables are destroyed only when the kernel finishes its execution.

CHAPTER 4

GPU BASED LITHOGRAPHY

In Chapter 2 lithography simulation using wavelets was introduced. It was found that by using a wavelet and by limiting the number of points where the aerial image intensity needs to be calculated the runtime was reduced resulting in a fast simulator. In this chapter the implementation of the wavelet based lithography simulator on a GPU is described.

4.1 Implementation

The first step in performing lithography simulation on a mask is to read in the description of the mask. The description contains all the metal layers, of which we simulate the second layer as it is the most dense for a given process technology. The other layers may be simulated similarly. The features on the mask are read into a data structure which divides the mask into grids, and then decomposes the features in the grids into contours, rectangles and lines.

In the simulator implemented in [16] the simulation points are generated based on an algorithm, and the aerial image simulation is carried out for all the simulation points. Further, the simulation point was moved around until the intensity value was greater than the value of the contour edge. This implementation results in a loop containing a lot of branches and for the GPU implementation we want to minimize the number of branches. In order to do so, the entire mask is divided into pixels of 5nm size. The pixels which lie on the edge of the contours are termed primary simulation points. If we follow the

approach presented in [16], we would calculate the intensity value of these points and then based on this value and the edge intensity value we would select the pixel either to the left or right of the simulated pixel as the next simulation point. As mentioned earlier, this approach leads to branches. On the GPU we select a few pixels to the right and the left of the primary simulation point as the secondary simulation points. These points represent the possible location of the contour edge. By simulating both the primary and the secondary simulation points, we can easily determine the final edge of the contour. The original contour and the simulation points are shown in Figure 18 in which the points in black are the primary simulation points while the ones in green are secondary simulation points.

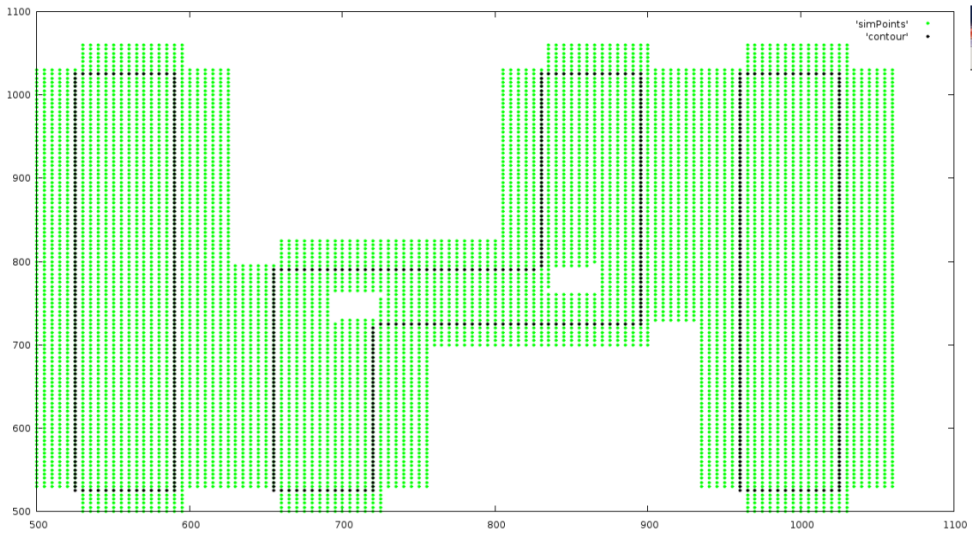

Figure 18. Simulation points

Once the data structure is populated we can begin to create pixels in the grid; each pixel has a dimension of 5nm. Initially the pixels are blank; later, we map the contours in the grid to the pixels. If a pixel is a part of a contour, then it has a value of 0, else it has a value of 1. In the process of multiplication and addition, a 0 will decrease intensity while a 1 will increase intensity. The simulation can proceed after all the values

are assigned. For every simulation point, a tile is created which represents the optical diameter. This tile is a 2D matrix of pixels. The optical diameter is 1µm in size and so the matrix has 200 points in each row and column (1µ divided by 5nm).The wavelet is also a matrix of the same size as the optical diameter. To find intensity, the corresponding elements of the two matrices are multiplied and all the values are summed up. This gives us the final intensity value.

Calculating the intensity of the simulation point is the most computationally intensive part of the simulation and it also needs a large memory bandwidth as we need to access 40000 elements thrice. The first access is to read the contour values, next to read the wavelet values and finally to write the intensity values, so this part of the simulation is executed on the GPU.

The best approach to saturate the GPU would be to use the device memory and transfer as large a part of the mask as possible, while retaining space for storing intensity values and the wavelet. This led to the use of grids. Each grid has 10000 nm^2 area as this is the maximum size that can fit on the device memory at a time. Once the mask has been divided into grids, one grid at a time is transferred to the device. The wavelet matrix is also copied and space is allocated for the final intensity values. After the computation is complete, we only need to copy the intensity values from the device to the host memory and update them in the mask data structure. The wavelet matrix cannot be retained for following kernel calls as the device does not guarantee the validity of the data structure over multiple calls, so we need to transfer this matrix for every grid. The other motive to keep the grids size as large as possible is to amortize the memory transfer overhead over as many pixels as possible.

This also has the advantage of being scalable for very large mask sizes. For large masks if we attempt to store the entire mask in the device memory, we would run out of

memory. But by dividing the mask into grids only sections of the mask are simulated at a time and stitched together later.

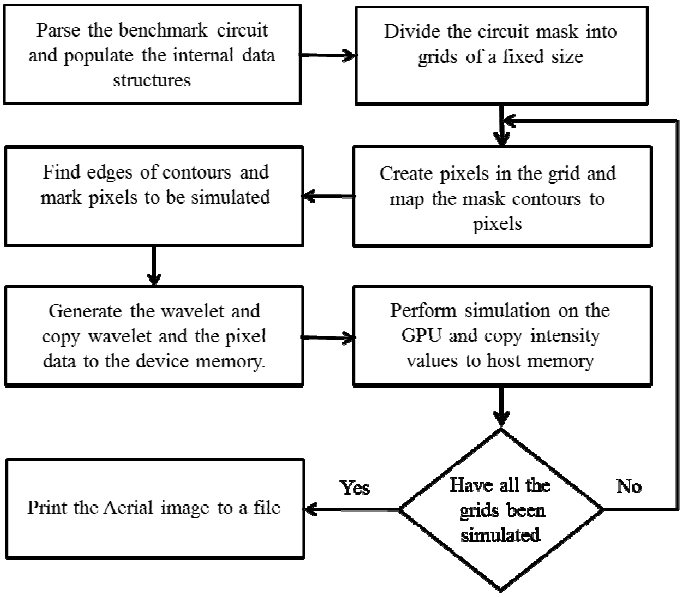

Figure 19. Flowchart

```
Procedure: performLithoSimulationOnCPU(){
   for(every pixel in the grid){
     if(pixel is a simulation pixel){
       multiply the contour and tile matrices;
       Sum all the terms in the product matrix;
       update intensity values in intensity matrix;
     }
   }
  }

Procedure: performLithoSimulationOnGPU(){
  Index = blockIdx.x * blockDim.x + threadIdx.x;
    if(index < totalpixels && shouldsimulatepixel){
      intensity= 0;
      for every element in the wavelet matrix{
        multiply the wavelet vale and the contour pixel;
        intensity += product;
      }
      intensityMatrix[index] = intensity;
    }
  }
}
```

Figure 20. CPU and GPU pseudo code of the methods used to perform simulation

After the calculation is complete for the entire grid, and the intensity value of all pixels has been obtained, we identify all the points that have an intensity value below the required aerial image intensity value. These points represent the edge of the aerial image. This process is repeated for every grid in the mask. As this part of the code is branch heavy it is executed on the CPU. The flow chart of the code can be found in Figure 19 and the pseudo code of these methods can be found in Figure 20. The aerial image for an example circuit is shown in Figure 21. The original contours are shown in black while the aerial image is shown in green.

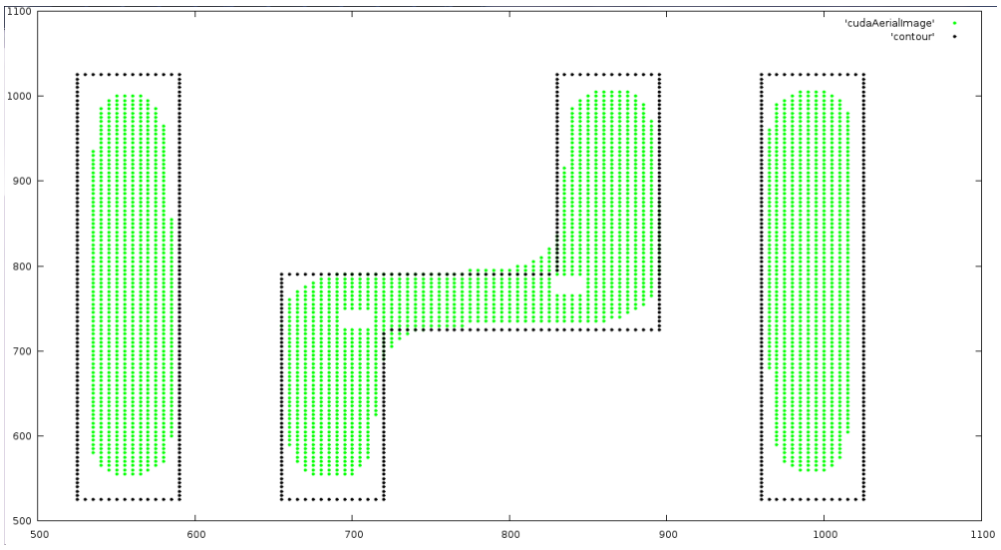

Figure 21. The figure shows the original mask and the aerial image obtained by our method

4.2 Results

The simulation was performed on a 2.6 GHz Core2Duo dual core machine with 4GB RAM, running Ubuntu 8.10. The GPU used was the Tesla C870, it belongs to the G80 architecture and has 128 cores with each core clocked at 1.35Ghz.The benchmark circuits used are from the ISCAS'85 benchmark suite. The results are plotted in Figure 22 and tabulated in

Table 1 . The circuits in the ISCAS 85 had a range of sizes with the smallest being c432 having 12 grids, each grid being $10\mu m^2$ to the largest, c6288 having 81 grids. As commercial circuits can be as large as $1mm^2$, the implementation has been designed to be scalable. When the circuit size doubles the number of grids increase by the square of the scale of the change. For example if the initial circuit size was $10\mu^2$ and the grid size was

also $10\mu^2$, there would be one grid. If the circuit size becomes $20\mu^2$, then there would be 4 grids. The circuits was twice as large and the number of grids increased by the scale of the change, this would lead to a quadratic increase in runtime.

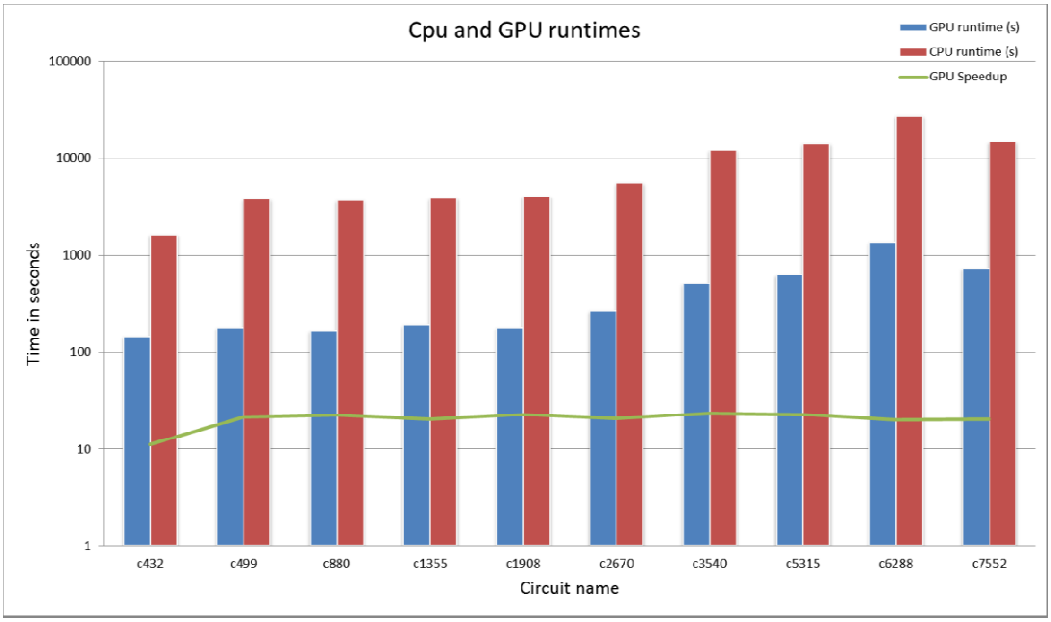

Figure 22. Chart showing the runtimes of CPU and GPU and the speedup

The plot shows us the runtimes of the GPU, CPU and the obtained speedup. We can see that an average speed up of 20x has been obtained for the benchmark circuits and the speedup is quite consistent across all the circuits. We can also see that the speedup is scalable with mask size which indicates that this method is suitable for use on very large production masks. The pixel simulator was implemented on the CPU in order to get the CPU runtimes, and so the GPU implementation is compared to the CPU implementation on the same platform. Our implementation is also faster than the original implementation presented in [16]. There is no other work involving use of GPU for lithography alone, although in [32] the authors use GPU for OPC. In [33] the authors present a FPGA accelerated lithography simulator, in which a sample mask of 200µm by 200µm is

simulated the authors report only the resulting speedup and not the absolute runtimes. C6288 is similar in size and takes about 7 hours with our GPU.

Table 1. CPU and GPU runtimes and speedup

CIRCUIT NAME	GPU TIME (S)	CPU TIME (S)	SPEEDUP
c432	144	1607	11.15
c499	176	3749	21.30
c880	164	3651	22.26
c1355	187	3824	20.45
c1908	176	4003	22.74
c2670	267	5482	20.53
c3540	504	11833	23.47
c5315	616	14081	22.85
c6288	1358	27339	20.13
c7552	721	14647	20.31

CHAPTER 5

OPC USING WAVELETS

There are two types of algorithms in model based OPC, polygon based and pixel based. In polygon based OPC the mask patterns are divided into regular polygons and OPC is carried out by shifting line segments until the final pattern is close to the required pattern. In pixel based OPC the mask is divided into pixels and the values of the pixels are modified to correct the mask. Our methods use model based OPC and the pixel paradigm.

5.1 Basic pixel based OPC

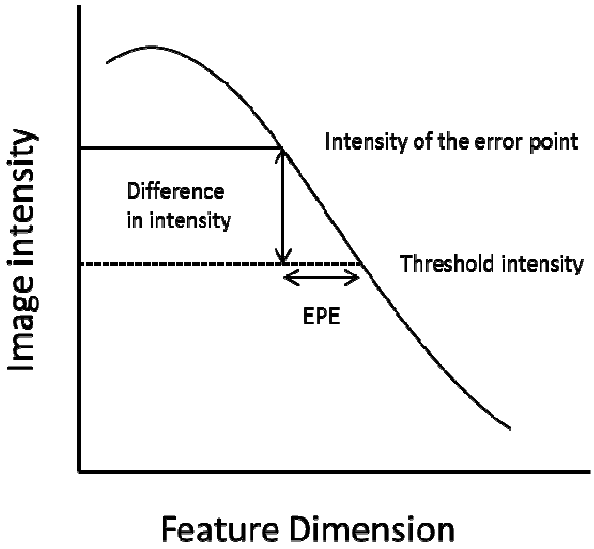

Figure 23. Difference in intensity for an error point

The first step in performing pixel based OPC is to divide the entire mask into pixels, where the pixels have different values based on whether the pixels belong to contours on the mask. Once the values have been assigned the intensity value of all

contour points and points around the contours is calculated. The edges of the aerial image are then determined by comparing image intensity of the points with the threshold intensity determined by the constant threshold model as explained in the lithography simulation chapter. Due to proximity effects the intensity of the points at the edge of the original contours differs from the threshold intensity. The threshold intensity determines the location of the contour, so the edge of the printed pattern moves away from the desired pattern due to this difference in intensity. This difference manifests itself as EPE as shown in Figure 23.

The aerial image is the starting point for the OPC algorithm. All the pixels which deviate from the expected location are termed as error points, and the intensity of these points must be corrected so that their intensity is less than or equal to the threshold intensity value. There are two kinds of error points; bridging and open, if the contour of the final patter is outside the edge of the original contour these points are termed as bridging points, if the final contour edge moves inward these points are called open points.

To correct the intensity of a tile whose size is equal to the optical diameter, the values of the pixels in the tile are modified and the intensity recalculated after each change. If the change in intensity is in the expected direction (the intensity of an open point should be decreased while that of a bridging point should be increased), then the change is retained, else it is discarded and the next pixel is chosen. The pattern of selection of the pixels also plays a large part in the quality of the final mask. The pixels are chosen in a radial direction around the error point in the implementation used in this thesis. This confines the changes to the region surrounding the error pixel and minimizing

the impact on the other features. By repeating this process until the intensity of the error pixel matches the threshold intensity, the original mask pattern is corrected. The flowchart of this process is presented in Figure 24.

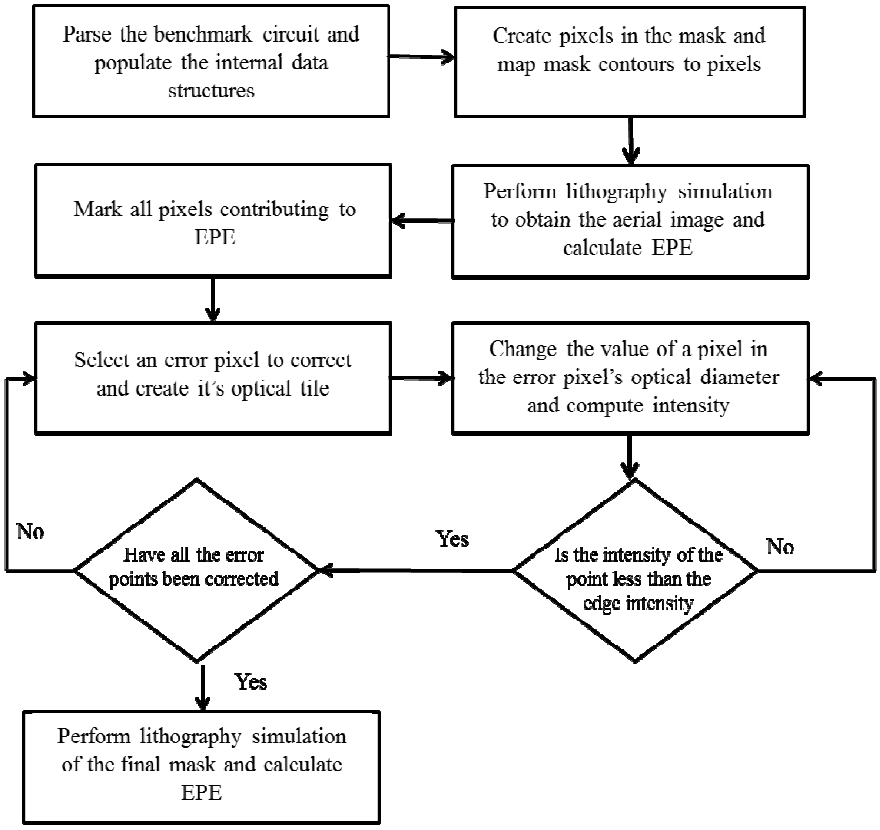

Figure 24. OPC flowchart

5.1.1 Scalability of the algorithm

The runtime of the implementation depends on the number of error points in the grid, the number of grids and the number of patterns in the design. On comparing two circuits with similar densities with the second being double the size of the first, the second circuit will have a runtime that is four times of the first. This means that the algorithm is of the order N^2 i.e. O (N^2), where N is the ratio of the area of the two circuits.

The runtime can be decreased by using GPU with a larger number of cores or by using multiple GPUs in parallel. Due to the nature of the CUDA API where there shouldn't be a dependency between two thread blocks because the order of execution of blocks is not guaranteed; the same code can be executed on different GPUs without any modifications. But this decrease is not necessarily linear due to limited bandwidth available on the GPU.

5.2 OPC with pattern matching

The mask contains rectangular shapes of different widths and the contours always have right angle corners. Although the number of different shapes is large, a commercial mask contains millions to billions of shapes and it is inevitable that the shapes repeat. We can take advantage of this fact to reduce our computation. During lithography simulation, a signature is calculated for all tiles. Each time an error pixel is selected to be corrected, its signature is compared to the signatures of previously corrected pixels. These signatures along with the coordinates of the error point are stored in a heap to allow fast comparison. When the tile for the next pixel is created its signature is matched with that of the earlier tiles, and if a match is found then the tile of the pixel whose coordinates are stored with the signature is copied to the tile of the current error point. We note that the tile of the matching point now contains the final pattern (after OPC). This way we can save on computation for a matching point. The flowchart of this process is shown in the Figure 25.

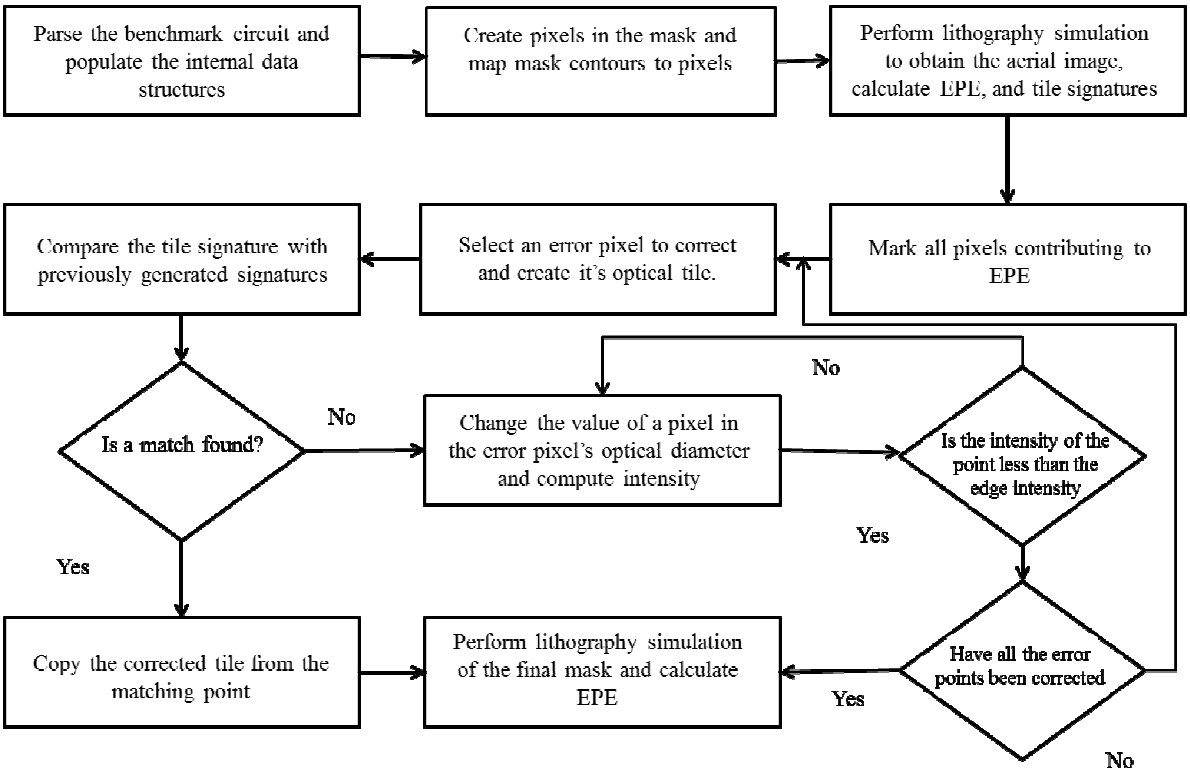

Figure 25. Flowchart for pattern matching OPC

5.3 Improved intensity calculation

As explained earlier, to perform OPC the intensity of an error pixel is corrected by changing the value of pixels within the optical diameter of the error pixel. The intensity of the error pixel is recalculated each time a pixel is modified. In the basic OPC method the intensity is recalculated by multiplying all the pixels in the optical diameter with the wavelet. The change in the intensity of the error pixel is limited to that contributed by the modified pixel. By calculating the intensity contribution of this pixel, and subtracting it from the intensity of the error pixel, we remove this pixel from the intensity calculation.

After the pixel value has been changed and its new intensity contribution calculated, the intensity of the error pixel is updated. This process decreases the number of calculations required to update the intensity of the error pixel.

5.4 Process Variation

In a regular OPC approach the correction is performed at a single focus and dosage. But there are variations in circuit manufacturing leading to variation in focus and dosage in addition to other parameters. A robust tool should be able to take process variation into account. There has been previous work in performing OPC with process variation in [34] and [35]. In [36], the authors present the concept of Process Window Optical Proximity Correction (PWOPC) which ensures high yield in addition to fulfilling the standard OPC objective of improving the printed pattern.

In this implementation of process variation the effect of focus variation is modeled by changing the scale of the wavelet. Various values of scale have been found to simulate focus variation in the lithography process. These values have been calibrated with the help of commercial lithography tools. The dosage variation behavior is captured by change the required edge intensity value obtained from (2) in section 2.3. A larger dose will increase this required value while a lower dose will decrease it. Again these values have been calibrated with commercial tools.

By incorporating these changes in the lithography model, aerial images can be obtained at various process corners. By performing OPC at these process corners we can correct masks to ensure good printability at any process corner.

5.5 Minimizing Shot size

The previous schemes have focused on the quality of the final pattern; this has led to patterns that have a minimum jog size of 5nm. A jog is the size of the smallest feature that can be added or subtracted from the original mask. Small jog sizes provide lower EPE at the cost of increased difficulty in manufacturing the mask while larger jog sizes reduce the mask manufacture cost at the expense of EPE [37]. Gupta et al describe the weight of different parameter in mask cost in their paper [38]. A good OPC tool should balance these two aspects of lithography.

When we use larger jog sizes, there are fewer opportunities for us to improve the intensity value of the error pixel. This is due to the fact that the optical diameter has a limited area and if we use larger regions at a time there are fewer locations to modify. Due to this reason the region to be modified must be carefully selected. This is done by following the pixel weights based method described below

- Select a pixel in the optical tile and change its value. Compute the intensity of the error point.
- If the change in intensity is beneficial to the intensity value, increase the weight of the pixel by 1 and revert back the change to the value of the pixel
- Repeat this process for all pixels in the tile.
- Repeat for all error pixels in the mask.

At the end of this process, all the pixels in the mask have weights; the pixels that have a large value of weight have the most beneficial effect on the error pixels. These pixels are now committed and the aerial image is now calculated. The entire process is now repeated until the EPE reaches an acceptable value. More strategies to reduce the mask cost are presented in [39].

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter we present the simulation results for OPC. The OPC simulation was performed on a 2.6 GHz Core2Duo dual core machine with 4GB RAM, running Ubuntu 8.10. The GPU used was the Tesla C870; it belongs to the G80 architecture and has 128 cores with each core clocked at 1.35 GHz. Some of the patterns used were taken from other papers related to OPC and others were taken from sections of Iscas'85 benchmark circuits.

6.1 Basic OPC

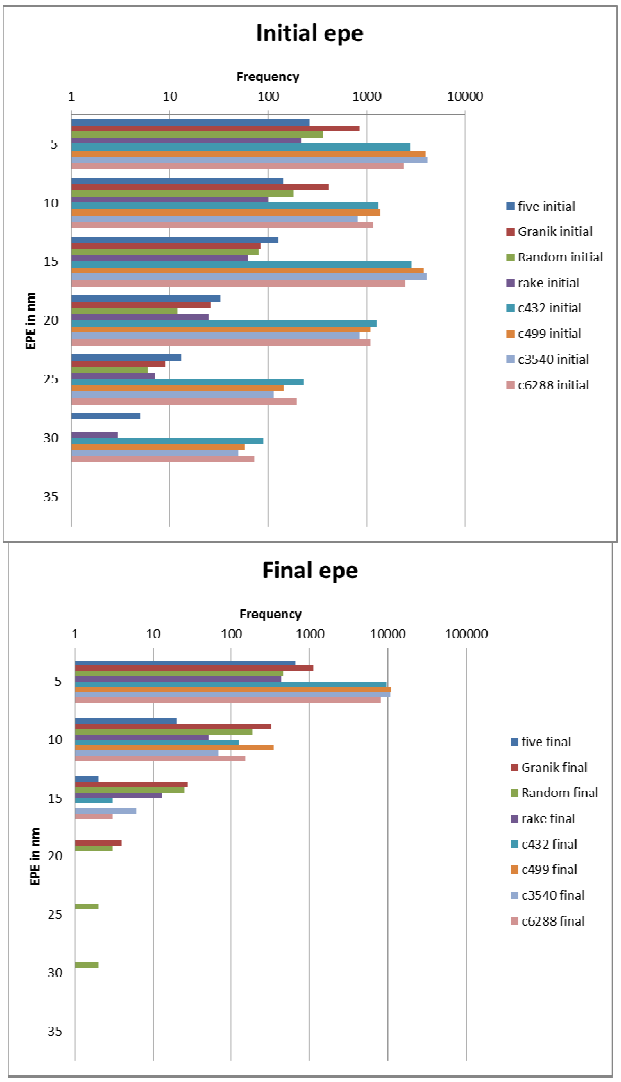

Figure 26. Histograms of Initial and final OPC values

Figure 26 shows the distribution of initial and final EPE. The figure on the left indicates the number of locations that have the specified EPE for a given circuit, while the figure on the right shows the distribution of EPE after OPC has been performed. It can be seen that the number of points having a large EPE has been reduced which bears testimony to the effectiveness of our method. The runtimes of the tool for various

circuits, the initial and final average EPE values and also the worst case EPE before and after OPC are presented in Table 2.

Table 2. OPC results

Circuit name	Initial average EPE(nm)	Final average EPE(nm)	Worst case EPE before OPC	Worst case EPE after OPC	Runtime (s) OPC alone	Runtime (s) OPC and Lithography
Five	9.81	5.12	30	15	4	11
Double rake	9.12	5.81	30	20	4	11
Granik	7.52	6.33	25	25	8	19
Random	8.11	6.95	30	30	4	11
C432	12.15	5.06	30	15	88	175
C499	11.31	5.15	30	10	89	176
C3540	11.09	5.03	30	15	83	171
C6288	12.12	5.09	30	15	73	158

An example circuit is shown in Figure 27, it shows the aerial image before and after OPC. The figure on the left shows several regions of line end shortening, all those regions are corrected in the figure on the right. The original and final masks corresponding to the aerial images are shown in Figure 28.

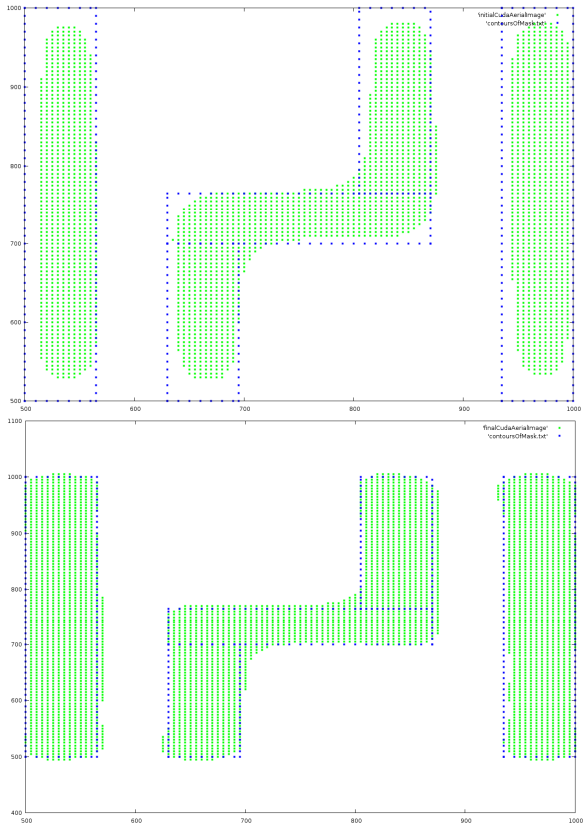

Figure 27. Aerial image before and after OPC

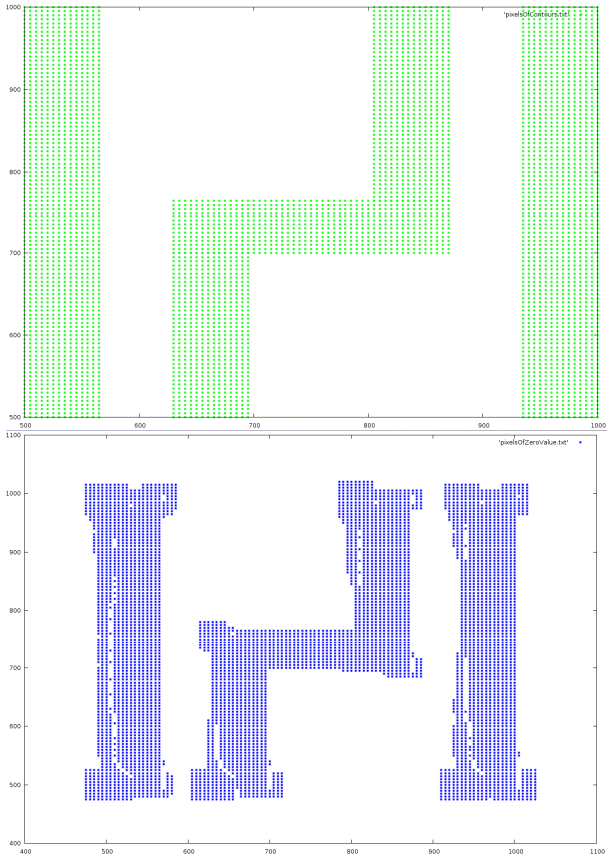

Figure 28. Mask before and after OPC

6.2 OPC with pattern matching

This section presents the results for OPC with pattern matching. The values for EPE are the same as in the basic method, with the changes being restricted to runtime alone. These results are presented in

Table 3. Runtimes for OPC with pattern matching

Circuit name	Runtime without pattern matching (s)	Runtime with pattern matching(s)	Speedup	Number of error points	Number of matching points
Double lines	5	2	2.5	1500	750
C432	88	74	1.18	26490	4260
C499	89	53	1.67	27199	5110
C3540	83	67	1.23	25833	5110
C6288	73	55	1.32	22300	5703

The first case consists of two contours, where the second contour is an exact match of the first one; this case represents the best speedup that can be achieved as all the error points of the second contour are identical to the first contour. As regular circuits will not have such a large overlap of contours, the speedup observed for the benchmark circuits will be lower. But on the whole an average speed up of 25% is observed. This method will be of particular use in masks which are used to manufacture highly regular circuits like FPGAs and RAM.

6.3 Improved intensity calculation

Table 4. Runtimes for OPC with improved intensity calculation

Circuit name	Basic OPC Runtime (s)	OPC Runtime with improved intensity calculation (s)	Speedup
Five	4	1	4
Double rake	4	1	4
Granik	8	2	4
Random	4	1	4
C432	88	19	4.6
C499	89	20	4.45
C3540	83	18	4.6
C6288	73	16	4.6

Table 4 presents runtimes for OPC using the improved intensity calculation method. A minimum speedup of 4x is observed for all the circuits. The benchmark circuits have a speedup greater than 4x. It can also be seen that these runtimes are better than those obtained from pattern matching

In [31] the authors indicate a runtime of 4.14s for an inverter at 65nm which has a size of $1um^2$. This can be compared to the double lines mask whose size is $1.6um^2$. It can be seen that our approach is faster than the one presented in [31]. In [25] the authors present a GPU based OPC technique which takes 0.11 hours for a $1mm^2$ chip. This runtime was obtained on a cluster of machines which contained 2 Intel quad core CPUs and 8 NVIDIA GTX 295 GPUs; each GPU having 480 cores giving us a total of 3840 cores. For reference our largest circuit is 5um by 5um and OPC takes 16s on a GPU having 128 cores.

6.4 OPC and process variation

Table 5. Results for OPC with process variation

Dose\Defocus	-5	0	+5
-10	10.25/5.19nm	10.27/5.22nm	10.25/5.27nm
0	11.18/5.12nm	11.29/5.16nm	11.32/5.17nm
+10	15.11/6.22nm	15.15/6.22nm	15.23/6.22nm

Table 5 presents the results for OPC with process variation for the section of the c499 benchmark circuit. The values of dose and focus are changed and the resulting initial and final EPE values have been reported for these combinations. The first column on the left represents the dose values while the first row represents values of focus. The results are shown in Figure 29 where it can be seen that our OPC tool can correct the mask for different process windows.

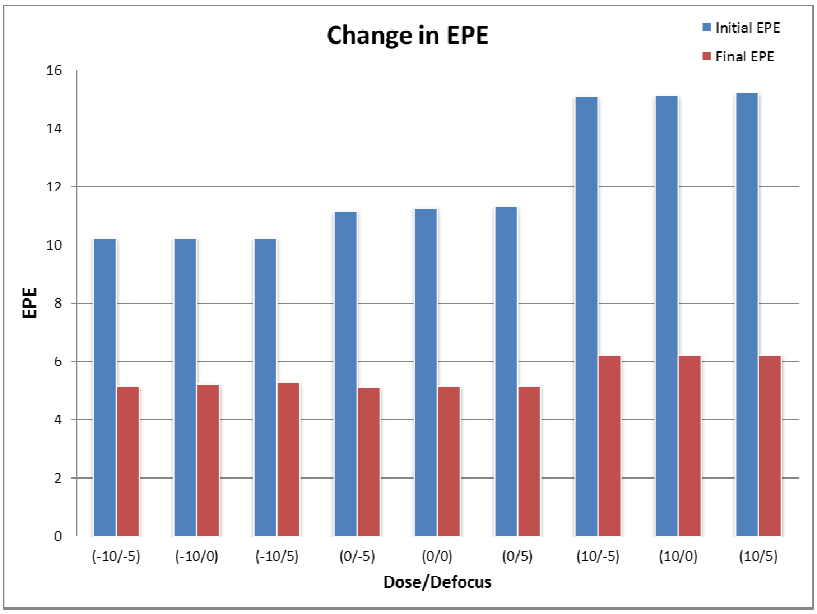

Figure 29. Process variation

6.5 Reducing shot size

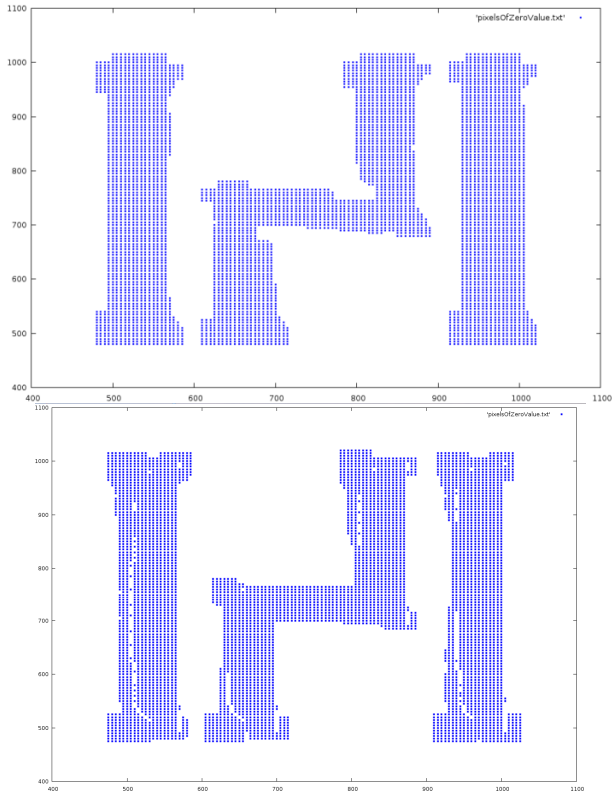

Figure 30. Comparison of final mask with and without shot size reduction

By controlling the size of the minimum feature that is changed at any given time we can decrease the mask manufacture cost. But as the feature size increases, the quality of the printed pattern is lowered resulting in larger EPE. The difference between a corrected mask which has larger shot sizes and one with a smaller shot size can be seen in Figure 30. The EPE values and the runtimes of this method are presented in

Table 6. The runtimes are larger than the basic OPC implementation due to the pixel weight based approach used in correcting the mask, also due to larger shot sizes the EPE is higher than that obtained by the basic OPC method.

Table 6. Results for OPC with shot size reduction

Circuit name	Initial EPE (nm)	Final EPE (nm)	Final EPE no shot minimize (nm)	Runtime (s)
Five	9.87	5.48	5.12	4
Double rake	9.122	5.98	5.81	3
Granik	7.50	5.86	6.33	6
Random	8.18	5.58	6.95	3
C432	12.15	6.69	5.06	75
C499	11.29	6.78	5.15	76
C3540	11.09	6.56	5.03	72
C6288	12.06	6.49	5.09	62

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis a GPU-based implementation of lithography simulation which is faster than existing simulators has been presented. Dividing the entire mask into pixels and minimizing the number of branches in the code has led to maximum utilization of resources on the GPU and decreased runtime. An average speed up of 20x compared to the CPU implementation and that the simulator is able to handle circuits of various sizes, has been demonstrated.

Several implementations of OPC which focus on key metrics of mask design and pattern quality have been shown. The OPC tool provides us with a mask pattern which generates patterns on the wafer very close to the desired pattern with minimum iterations. The quality of the output as well as the runtime is better than other tools. An improved intensity calculation scheme has been implemented and shown to reduce runtime. The adaptability of the OPC tool has been demonstrated by its use in correcting masks under different process corners. A key concern of pixel based OPC correction has been addressed by the implementation of shot minimization. This improves the printability of masks by increasing the size of the smallest mask feature which decreases the cost of mask manufacture.

A paper titled “Detecting shorts and open faults in a mask using lithography simulation” has been accepted in North Atlantic Test Workshop 2010. In this paper, the GPU based simulator was used to evaluate faults in the printed pattern. Based on the work presented in this thesis a paper titled “GPU accelerated lithography using wavelets” is under review in ISQED 2011.

APPENDIX

TEST PATTERNS

The test patterns used to evaluate the OPC tool are presented below.

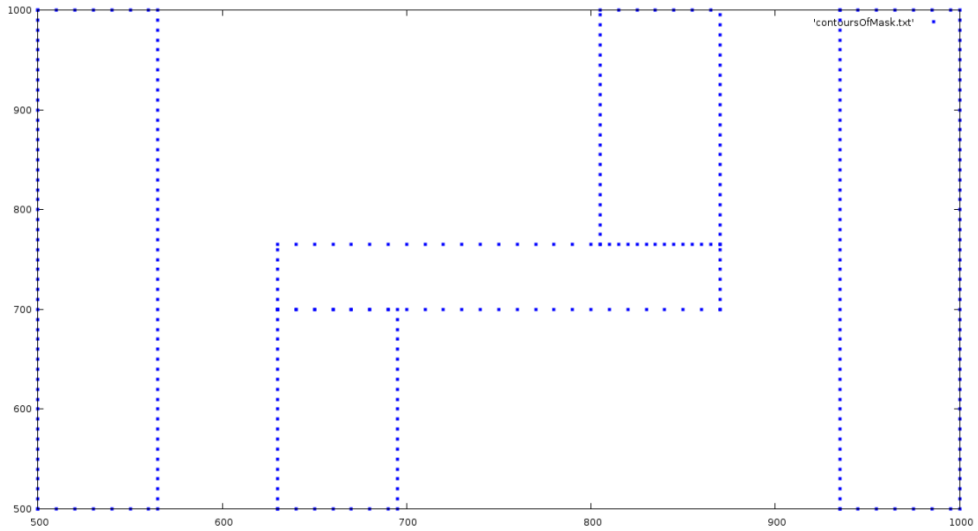

Figure 31. Five

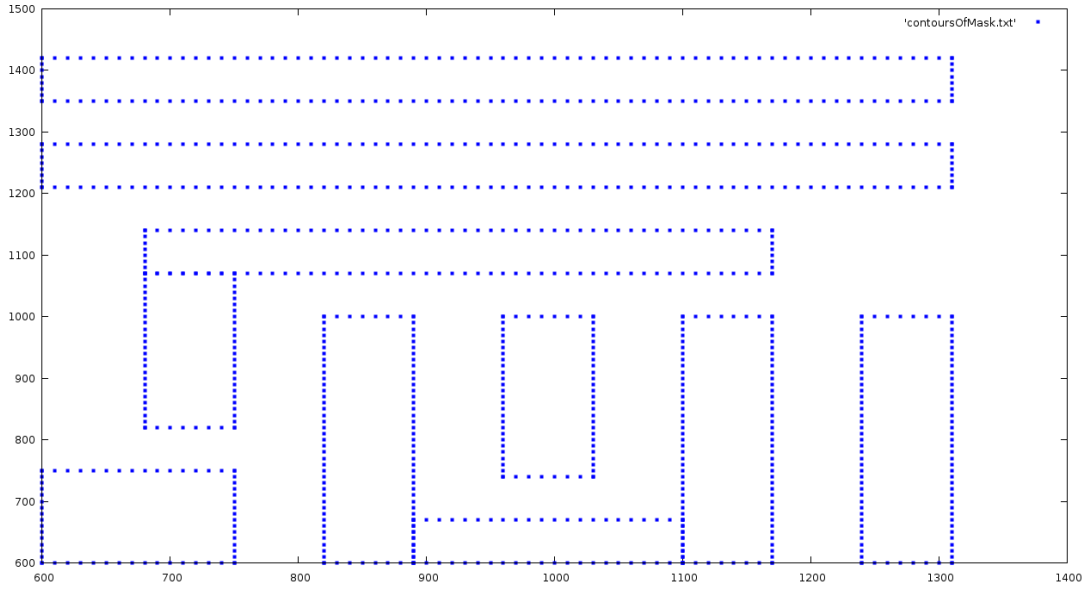

Figure 32. Granik

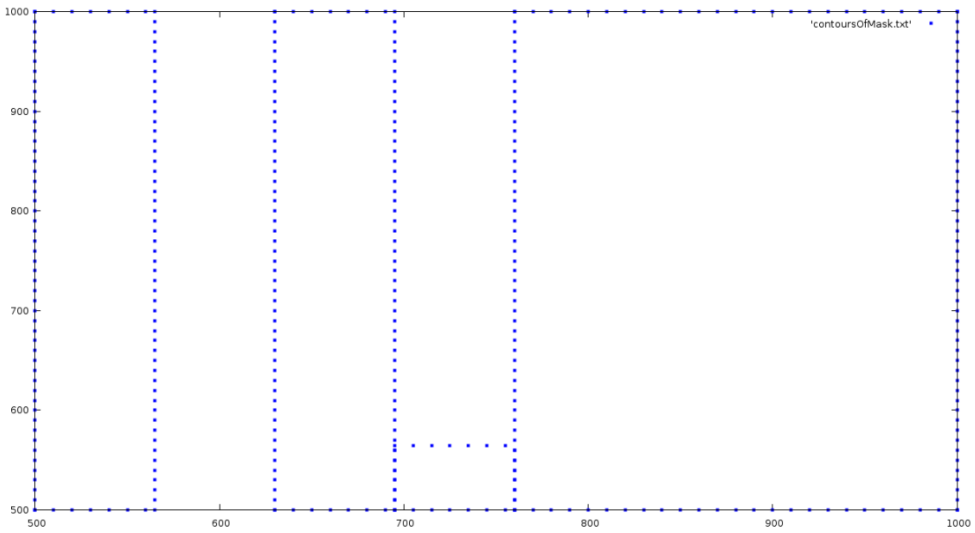

Figure 33. Random

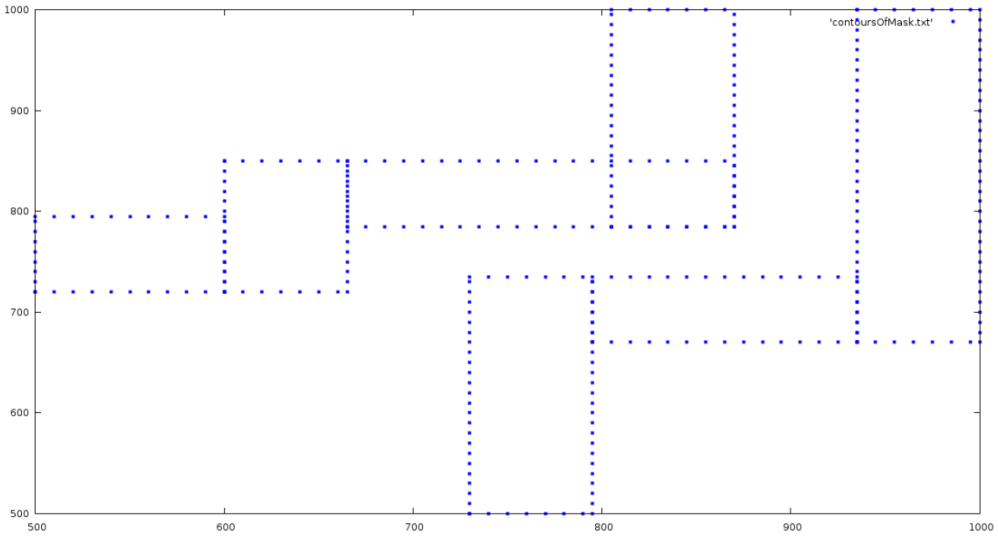

Figure 34. Double Rake

BIBILIOGRAPHY

[1] Rothschild, M et al, “Recent trends in optical Lithography”, in Lincoln Laboratory journal, Vol 14, No 2, 2003

[2] Chris A Mack, "Fundamental Principles of Optical Lithography", ISBN 9780470727300

[3] Mitra, J.; Peng Yu; Pan, D.Z.; , "RADAR: RET-aware detailed routing using fast lithography simulations," Design Automation Conference, 2005. Proceedings. 42nd , vol., no., pp. 369- 372, 13-17 June 2005

[4] Peng Yu, David Z. Pan and Chris A. Mack, "Fast Lithography Simulation under Focus Variations for OPC and Layout Optimizations", in Proceedings of the 43rd annual Design Automation Conference, 2006, Pg 785 – 790

[5] N. B. Cobb. “Fast optical and process proximity correction algorithms for integrated circuit manufacturing” PhD thesis, UC Berkeley, 1998.

[6] David B Kirk, Wen Mei w. Hwu, “Programming Massively Parallel Processors- A Hands on Approach,” ISBN 0123814723

[7] NVIDIA CUDA Programming Guide v3.0, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

[8] Yangdong Deng; Shuai Mu; , "The potential of GPUs for VLSI physical design automation," Solid-State and Integrated-Circuit Technology, 2008. ICSICT 2008. 9th International Conference on , vol., no., pp.2272-2275, 20-23 Oct. 2008

[9] Qihang Huang et al, “GPU as a General Purpose Computing Resource”, 9th International Conference on Parallel and Distributed Computing, 2008

[10] Owens, J.D.; Houston, M.; Luebke, D.; Green, S.; Stone, J.E.; Phillips, J.C.; , "GPU Computing," Proceedings of the IEEE , vol.96, no.5, pp.879-899, May 2008

[11] Yeung, Michael S., “Fast and rigorous three-dimensional mask diffraction simulation using battle-lemarie wavelet-based multiresolution time-domain method”. Anthony Yen, Ed., vol. 5040, SPIE, pp. 69–77.

[12] Wong, A.K.; Neureuther, A.R.; , "Rigorous three-dimensional time-domain finite-difference electromagnetic simulation for photolithographic applications," Semiconductor Manufacturing, IEEE Transactions on , vol.8, no.4, pp.419-431, Nov 1995

[13] Peng Yu, David Z. Pan and Chris A. Mack, "Fast Lithography Simulation under Focus Variations for OPC and Layout Optimizations", in Proceedings of the 43rd annual Design Automation Conference, 2006, Pg 785 – 790

[14] Frederick H Dill, ”Optical lithography”, IEEE transactions on electron devices, Vol ID 22, No 7, July 1975

[15] John Randall, Kurt Ronse, Thomas Marschner, Mieke Goethals, Monique Ercken, "Variable Threshold Resist Models for Lithography Simulation", in SPIE Vol. 3679 SPIE Conference on Optical Microlithography XII March 1999

[16] Rodrigues, R.; Sreedhar, A.; Kundu, S.; , "Optical lithography simulation using wavelet transform," Computer Design, 2009. ICCD 2009. IEEE International Conference on , vol., no., pp.427-432, 4-7 Oct. 2009

[17] Paul S Addison, “The Illustrated Wavelet Transform Handbook”, ISBN 0750306920

[18] N. B. Cobb and Y. Granik, “Model-based OPC using the MEEF matrix,” in Proc. SPIE 4889, Dec. 2002, pp.1281–1292.

[19] J. Stirniman and M. Rieger, “Fast proximity correction with zone sampling”, In Proceedings of SPIE Symposium on Optical Microlithography Vol 2197, pages 294-301, Santa Clara, 1994.

[20] J. Stirniman and M. Rieger, “Quantifying proximity and related effects in advanced wafer processes”, In Proceedings of SPIE Symposium on Optical Microlithography Vol 2440, pages 252-260, Santa Clara, 1995.

[21] J. Stirniman and M. Rieger, “Spatial-filter models to describe IC lithographic behavior”, In Proceedings of SPIE Symposium on Optical Microlithography Vol 3051, pages 469-478, Santa Clara, 1997.

[22] B. Painter, L. L. Melvin, III, and M. L. Rieger, “Classical control theory applied to OPC correction segment convergence,” in Proc. SPIE 5377, May 2004, pp. 1198–1206.

[23] W. C. Huang, C. M. Lai, B. Luo, C. K. Tsai, M. H. Chih, C. W. Lai, C. C. Kuo, R. G. Liu, and H. T. Lin, “Intelligent model-based OPC,” in Proc. SPIE 6154, Apr.2006, pp. 1065–1073.

[24] N. Cobb and Y. Granik, “New concepts in OPC,” in Proc.SPIE 5377, 2004, pp. 680–690.

[25] Ilhami Torunoglu, Ahmet Karakas, Erich Elsen, Curtis Andrus, Brandon Bremen and Pururav Thoutireddy, "OPC on a single desktop: a GPU-based OPC and verification tool for fabs and designers", Proc. SPIE 7641, 764114 (2010); doi:10.1117/12.846636

[26] J. Ye, Y.-W. Lu, Y. Cao, L. Chen, and X. Chen, "System and method for lithography simulation," Patent US 7,117,478 B2, Jan. 18, 2005.

[27] Peng Yu, Sean X. Shi and David Z. Pan, "True process variation aware optical proximity correction with variational lithography modeling and model calibration", J. Micro/Nanolith. MEMS MOEMS 6, 031004 (Sep 11, 2007); doi:10.1117/1.2752814

[28] Owens, J.D.; Houston, M.; Luebke, D.; Green, S.; Stone, J.E.; Phillips, J.C.; , "GPU Computing," Proceedings of the IEEE , vol.96, no.5, pp.879-899, May 2008

[29] Qihang Huang; Zhiyi Huang; Werstein, P.; Purvis, M.; , "GPU as a General Purpose Computing Resource," Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on , vol., no., pp.151-158, 1-4 Dec. 2008

[30] Peng Yu; Pan, D.Z.; , "A novel intensity based optical proximity correction algorithm with speedup in lithography simulation," Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on , vol., no., pp.854-859, 4-8 Nov. 2007

[31] Peng Yu , Sean X. Shi , David Z. Pan, "Process variation aware OPC with variational lithography modeling", Proceedings of the 43rd annual conference on Design automation, July 24-28, 2006, San Francisco, CA, USA

[32] Ilhami Torunoglu, Ahmet Karakas, Erich Elsen, Curtis Andrus, Brandon Bremen and Pururav Thoutireddy, "OPC on a single desktop: a GPU-based OPC and verification tool for fabs and designers", Proc. SPIE 7641, 764114 (2010);

[33] Jason Cong, Yi Zou, "FPGA-Based Hardware Acceleration of Lithographic Aerial Image Simulation", Sep 2009, Vol. 2, Issue 3, TRETS

[34] J. L. Sturtevant, J. A. Torres, J.Word, Y. Granik, and P. LaCour, "Considerations for the use of defocus models for OPC," in Proc. SPIE 5756, 2005, pp. 427–436.

[35] N. B. Cobb and Y. Granik, "OPC methods to improve image slope and process window," in Proc. SPIE 5042, pp. 116–125, 2003.

[36] A. Krasnoperova, J. A. Culp, I. Graur, S. Mansfield, M. Al-Imam, and H. Maaty, “Process window OPC for reduced process variability and enhanced yield,” in Proc. SPIE 6154, Apr. 2006, pp. 1200–1211.

[37] Ayman Yehia, "Mask-friendly OPC for a reduced mask cost and writing time", Proc. SPIE 6520, 65203Y (2007)

[38] Gupta, P.; Kahng, A.B.; Sylvester, D.; Yangt, J.; , "Performance-driven OPC for mask cost reduction," Quality of Electronic Design, 2005. ISQED 2005. Sixth International Symposium on , vol., no., pp. 270- 275, 21-23 March 2005

[39] Michael L. Rieger, Jeffrey P. Mayhew, Jiangwei Li and James P. Shiely, "OPC strategies to minimize mask cost and writing time", Proc. SPIE 4562, 154 (2002)